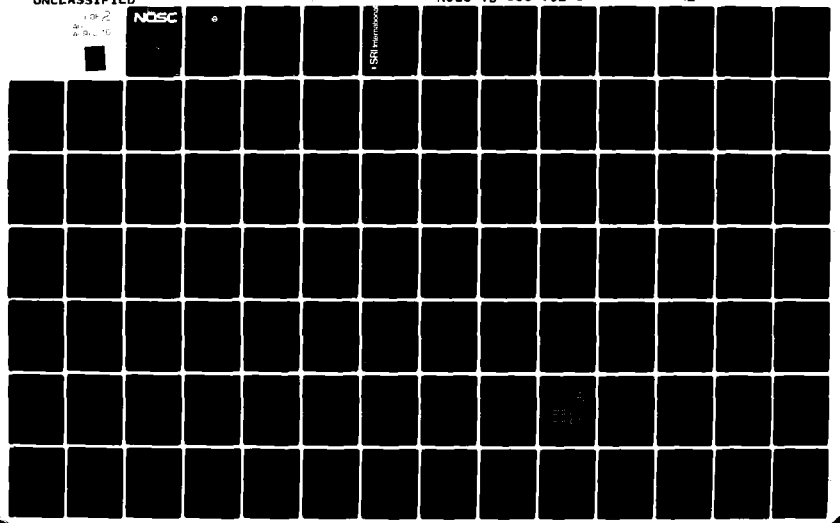
AD-A091 270 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK, VOLUME--ETC(U)
JUN 79 W L SUTTON, L ROBINSON N00123-76-C-0195
UNCLASSIFIED NOSC-TD-366-VOL-1 NL



LEVEL II 12 13

NOSC

NOSC TD 366

NOSC TD 366

Technical Document 366

THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK

Volume I: The Foundations of HDM

June 1979

Prepared for

Naval Ocean Systems Center

Approved for public release; distribution unlimited

NAVAL OCEAN SYSTEMS CENTER
SAN DIEGO, CALIFORNIA 92152

NAVAL OCEAN SYSTEMS CENTER, SAN DIEGO, CA 92152

AN ACTIVITY OF THE NAVAL MATERIAL COMMAND

SL GUILLE, CAPT, USN
Commander

HL BLOOD
Technical Director

ADMINISTRATIVE INFORMATION

The HDM Project was funded under Navy Element 62721N. The work leading to this publication was the result of NOSC Contract N00123-76-C-0195 with SRI International. The principal researchers were Dr. Karl Levitt (Volume III), Mr. Lawrence Robinson (Volume I), and Dr. Brad A. Silverberg (Volume II). The Navy Technical Monitor for the work performed under contract was W. Linwood Sutton, NOSC Code 8324.

Reviewed by
J. B. Balistrieri, Acting Head
C^3I Facilities Engineering &
Development Division

Under authority of
V. J. Monteleon, Acting Head
Command, Control, Communications
and Intelligence Systems Department

(18) NOSC (17) TD-366-Vol-1

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NOSC Technical Document 366	2. GOVT ACCESSION NO. AD-A091	3. RECIPIENT'S CATALOG NUMBER 270
4. TITLE (and Subtitle) THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK: Volume I: The Foundations of HDM		5. TYPE OF REPORT & PERIOD COVERED Technical document 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) W. Linwood Sutton (contract monitor) Lawrence Robinson		8. CONTRACT OR GRANT NUMBER(s) N00123-76-C-0195
9. PERFORMING ORGANIZATION NAME AND ADDRESS SRI International 333 Ravenswood Avenue Menlo Park, CA 94025		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62721N
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Ocean Systems Center San Diego, CA 92152		12. REPORT DATE June 1979 13. NUMBER OF PAGES 98
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Ocean Systems Center, Code 832 San Diego, CA 92152		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

abstract machines, abstraction, formal specification, hierarchical structure, Hierarchical Development Methodology (HDM), modules, software development process, software methodology, verification.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

HDM (the SRI Hierarchical Development Methodology) is an approach to software development that attempts to structure the overall development process by providing a unified framework that addresses most aspects of system development. Volume I of the HDM Handbook describes the basic concepts of HDM and shows how these concepts have been embodied in a computational model. It also presents procedures and guidelines for using them to produce a software system.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

410 221

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SRI International

THE HDM HANDBOOK

Volume I: The Foundations of HDM

Deliverable A006

SRI Project 4828
Contract N00123-76-C-0195

June 1979

By: Lawrence Robinson, Computer Scientist

Computer Science Laboratory
Computer Science and Technology Division

Prepared for:

Naval Ocean Systems Center
San Diego, California 95152

Attention: W. Linwood Sutton, Contract Monitor

CONTENTS

Foreword

This volume includes an introduction to all three volumes of the Handbook:

Volume I: The Foundations of HDM
Volume II: The Languages and Tools of HDM
Volume III: A Detailed Example in the Use of HDM

The author wishes to acknowledge the considerable assistance of Karl N. Levitt, Brad A. Silverberg, Peter G. Neumann, and Jack Goldberg in the preparation of this volume, as well as of HDM users such as Richard J. Feiertag and Tom Berson, whose feedback has greatly increased the applicability of HDM. The author is also indebted to Lin Sutton for his patience.

Lawrence Robinson is currently with The Ford Aerospace and Communications Corporation, Palo Alto, California.

I INTRODUCTION TO THE HDM HANDBOOK

The HDM Handbook is a three-volume tutorial on the Hierarchical Development Methodology (HDM), evolved at SRI International (formerly Stanford Research Institute) to aid in the production of correct, reliable, and maintainable software systems.

A. HDM

HDM is an approach to software development that attempts to structure the overall development process by providing a unified framework that addresses most aspects of system development. It extends and integrates many of the techniques proposed by Dijkstra, Parnas, Hoare, and Floyd.

HDM can be understood in terms of its computational model and its concepts, languages, and tools. It integrates the concepts of abstraction, hierarchical decomposition, modularity, and formal specification. These provide the basis for the computational model, consisting of abstract machines, data representations, and abstract programs, together with modular units for their specification.

Under HDM, the system development process is divided into seven stages, as follows.

* (Stage 1) Conceptualization of the system's requirements.
* (Stage 2) Selection of the user interface and the target machine.
* (Stage 3) Decomposition of the system into a hierarchy of components, that is, the formation of abstractions.
* (Stage 4) Specification of each component.
* (Stage 5) Representation of the data of each component in terms of the data of the component(s) at the next lower level.
* (Stage 6) Abstract implementation of the operations of each component.
* (Stage 7) Conversion of the abstract implementations into executable code.

Each of these stages adds structure and precision to the development process.

Of these seven stages, Stages 1-4 are part of the design process, and Stages 5-7 are part of the implementation process. These stages proceed in roughly sequential order, although considerable backtracking is common in the design stages. In addition, there are optionally associated with HDM a variety of stages relating to verification. For present purposes these may be thought of as a logical Stage 8, although the effort can be distributed as follows to provide verification during the design process, rather than merely at the end.

* (For Stage 1) Justification of the requirements.
* (For Stage 4) Verification of the consistency between specifications and requirements.
* (For Stage 6) Verification of the consistency between abstract programs and specifications.
* (For Stage 7) Verification of the consistency between executable programs and abstract programs.

There are various explicit languages used with HDM to express decisions made and details established during the design and implementation stages. These deal with the following issues:

* The structure of the hierarchical decomposition
* The intended behavior of each component, specified as an independent entity
* The relationships among the data in different components, i.e., relating abstract data objects to more concrete data objects
* The implementation decisions for each component, described as abstract or concrete programs

There are also tools used with HDM that help to determine if the structure, specifications, representations, and implementations are each syntactically well-formed in themselves, and consistent with each other.

The major benefits of HDM can be summarized as follows:

* It encourages the structuring of systems into components (each of manageable size), according to the principles of modularity

described by Parnas [12].

* HDM focuses attention on the properties of data: its abstraction, specification, and representation. This is vital in effectively designing large systems -- e.g., see Hoare [7].

* It encourages a designer to formulate systematically and to record precisely decisions made during system development [13]. Design decisions that have significant impact on the system and that would be difficult to change later are formulated early in the development process. Decisions that have only minor impact on the system are postponed.

* It provides tools that support the development of a system by alerting developers to inconsistencies and by performing many detailed bookkeeping chores.

HDM is currently being used in the development of several experimental systems (e.g., [10], [3], [19], [4]) and two production systems, KSOS [9] and its companion system KSOS-6. Implementations for these systems are currently planned or in progress. All but one of these systems involves design verification, and program verification is expected to be undertaken in most of them.

Nevertheless, HDM is not yet complete. Most importantly, a formal semantics for only a subset of HDM's specification language (SPECIAL) currently exists [1]. Work continues on formalizing all of SPECIAL.

In addition, the tools to support Stages 6 and 7 are currently under development (although the language for expressing implementation decisions has been completed and is used in this handbook). Many additional tools that are not described in this handbook also exist, notably those relating to verification. These include tools for verifying the consistency of specifications with a formal model for multilevel security, and tools for verifying program-specification consistency.

HDM is continuing to evolve, and its evolution will continue as more is learned -- through its use -- about its applicability in the software development process. Thus, HDM is simultaneously a facility for software production and a vehicle for research. In particular, difficult topics concerning performance analysis and parallel processing are being studied. Future versions of this handbook will incorporate

changes in HDM that result from its continuing evolution.

B. The Structure of the Handbook

1. Volume I: The Foundations of HDM

The work of Dijkstra, Parnas, Floyd, and Hoare has produced several powerful concepts for improving the quality of software systems. These concepts have been integrated and then embodied in a computational model that forms the basis of HDM. Volume I describes these concepts and the computational model, and presents procedures and guidelines for using them to produce a software system. The presentation is detailed, but is intended to be understandable to those who are moderately familiar with system design.

2. Volume II: The Languages and Tools of HDM

Volume II provides a self-contained description of the features of the languages of HDM and its main on-line tools, from the viewpoint of a user of HDM. It also contains a short summary of the ideas presented in Volume I.

3. Volume III: A Detailed Example in the Use of HDM

HDM has been created for use in the development of real (and therefore, unfortunately, large) software systems. Thus a true understanding of its operation and an appreciation of its benefits are difficult to achieve without an illustration of its application to the development of a large software system. The example chosen for this volume (a system that computes a frequency table of words in a file) is large enough to illustrate the capabilities of HDM, although it is not as complex as some systems (as noted above) currently using HDM. The example system is developed step-by-step, the languages of HDM are presented, and the decisions actually made during system development are described.

C. Other References

A valuable though slightly out-of-date guide to the languages and tools of HDM is given in "The SPECIAL Reference Manual" [18]. An overview of HDM suitable for project managers is found in "HDM -- Command and Staff Overview" [16].

II INTRODUCTION TO VOLUME I

HDM provides an integrated collection of languages and tools that aid in the software development process. HDM addresses many of the aspects of the general software problem -- namely that software is often late, too costly, unreliable, and noncompliant with its requirements.

In an attempt to address the software problem, researchers have advanced many concepts that can be applied to achieve quality software. These concepts take the form of desirable properties, design techniques, guidelines, and procedures. However, the mere presence of these concepts is no guarantee that the quality of software can be improved. Although there have been some benefits from this intellectual ferment, these guidelines have largely failed to improve the quality of production software because they have been piece-meal attempts rather than comprehensive solutions.

In developing HDM, we have selected some particularly useful concepts and integrated them into a unified approach that encourages software developers to think about software development in terms of these concepts. This approach defines a system as consisting of a set of components arranged in a particular structure. The components are specified using languages developed for that purpose. Some properties of the specifications can be evaluated by on-line tools; others can be measured by subjective evaluation. The languages and tools of HDM have been designed to enforce its concepts and to realize its mechanisms.

This volume describes the basic concepts of HDM. The stages provide a suggested ordering of system development. Guidelines for the use of HDM are also presented.

III THE CONCEPTS OF HDM

This chapter describes the following concepts that serve as a basis for HDM. These concepts are based largely on ideas first proposed by Dijkstra, Parnas, Floyd, and Hoare.

* Abstraction of procedure and data
* Hierarchies of abstract machines
* Appropriate modularity
* Specification of modules
* Verification of design and of implementation
* Data representation

The chapter then presents a model for describing the structure and interdependence of decisions, and concludes with a description of HDM's approach to the integration of these concepts.

A. Abstraction

Abstraction is potentially the most powerful technique available to system designers, having been used by mathematicians and scientists for centuries. There may be many valid ways to look at a large system, each way approriate in a particular context. Abstraction is the process of isolating just those properties of a system interface that need to be visible in order to explain that interface or to understand it more easily. An abstraction of a system is a set of properties that can be used in place of the system itself, under particular circumstances.

The use of abstraction in software has not been as fundamental nor as universal as is common in many engineering disciplines. This is primarily due to a lack of adequate methods for describing software abstractions and the failure to agree on a standard set of abstractions for software design.

We illustrate different kinds of abstraction with the example of a conventional flop-flop in engineering. The different usages of

abstraction are important in describing and understanding complex concepts and systems.

When we take previously defined or existing components, connect them together in "accepted" ways, and regard the assemblage as an indivisible unit, we have performed functional abstraction. In other words, we abstract certain operational properties of the individual components and assign these properties to the abstracted entity. The other direction, decomposing the abstracted object into functional components, can be considered an instance of the "divide and conquer" approach.

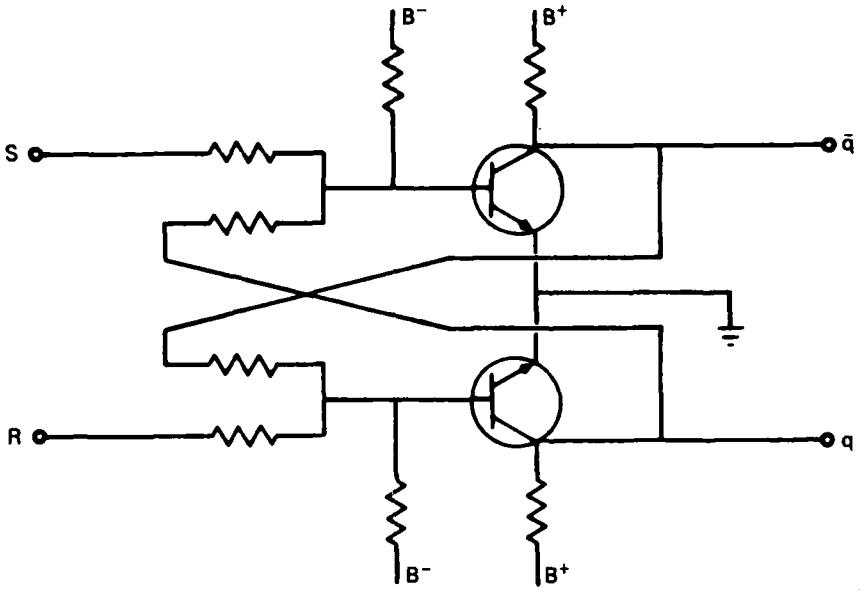
Functional abstraction relates to the operational behavior of an object. The result of the abstraction is a set of "higher-level" operations. Note that the input and output data processed by that object remain the same.

The definition of the object's behavior should be independent of any given implementation. An implementation is proved "correct" by demontrating that the externally observable operation of the implementation is precisely that given by the definition. Once the definition is formulated, however, only that definition may be consulted regarding the object's behavior; we may not look inside its decomposition and reason about the implementation. The abstraction acts like a "black box".

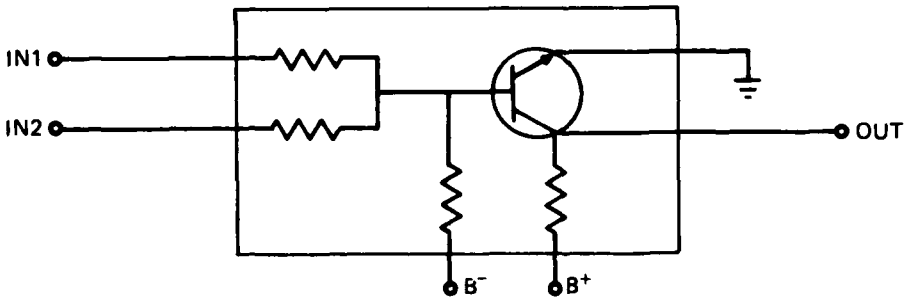
Two views of functional abstraction are presented for the flip-flop example. The first is a realization of a flip-flop in terms of discrete electronic components, e.g., transistors, resistors, capacitors, etc. -- see Figure III-1a. The second is a realization of a flip-flop via units called logic gates, which are in turn composed of discrete electronic components. A gate is shown in Figure III-1b, and the realization of a flip-flop in terms of gates is shown in Figure III-1c. The astute reader will have realized that the second view presents a hierarchy of abstractions.

Now, to understand the operation of the flip-flop (i.e., how the flip-flop transforms its data of voltages and currents), we derive its behavior from the known behavior of its components (operating on the

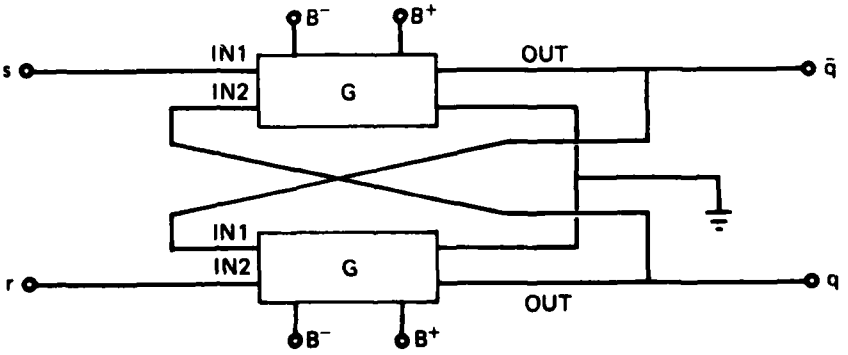
Figure III-1: Functional Abstraction



(a) A FLIP-FLOP IN TERMS OF PRIMITIVE COMPONENTS



(b) A GATE IN TERMS OF PRIMITIVE COMPONENTS



(c) FLIP-FLOP IN TERMS OF TWO GATES G

same data).

Note that in general, the difficulty of such an analysis depends on (1) how "abstract" are the components, and (2) how appropriate is the abstraction. For example, since (1) logic gates are more abstract than discrete components, and (2) the abstraction of "logic gates" is appropriate for flip-flops, we would expect the logic gate view to be more advantageous in the analysis of flip-flops.

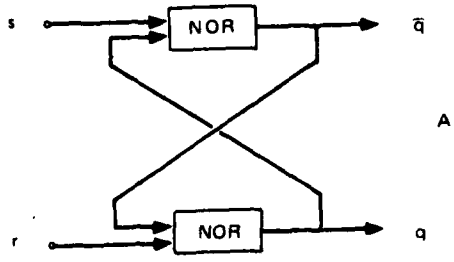
It is important to see that an abstraction also hides some of the properties of its components. For example, in our logic gate abstraction of a flip-flop, we analyze a flop-flop in terms of logic gate properties, and not discrete component properties, even though logic gates are composed of such discrete components. Thus, if we want our flip-flop to provide some capability not provided by logic gates, the logic gate abstraction is inappropriate.

When it is desired to build a flip-flop meeting certain voltage and current specifications, a configuration of possible components can be analyzed to determine whether or not it meets its specifications. One advantage of functional abstraction is that many of an object's components may be identical; it is necessary to understand the replicated component only once in order to understand all of its occurrences. Thus, we see that functional abstraction encourages the re-usage of known components, which in turn encourages the construction of a hierarchy of abstractions.

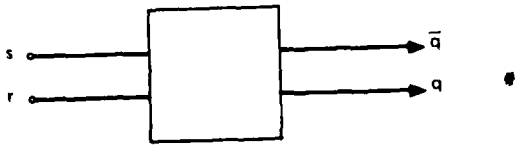
In representational abstraction the data operated on by the system or its components are conceptually replaced by simpler data having only a subset of the properties of the original data. For example, because a gate or a flip-flop has only two stable voltages at its inputs or outputs, the information in each input or output line can be represented as a single bit, representing high/low output. See Figure III-2a for the representational abstraction of a flip-flop composed of gates. Note that some data can vanish under representational abstraction. For example, the power supply and ground do not convey any information about the dynamic function of the flip-flop, so they have been eliminated.

Using this abstraction, a gate can be thought of as outputting a

Figure III-2: Representational Abstraction and Mathematical Models



(a) **A FLIP-FLOP USING BITS AND NOR GATES**
(representational abstraction)



#

s	r	OLD q	NEW q
0	0	1	1
0	0	0	0
1	0	X	1
0	1	X	0
1	1	X	?

WHERE $\bar{q}$ = NOT q
X = anything
? = indeterminate

(b) **A FLOW-TABLE DESCRIPTION OF A FLIP-FLOP**
(mathematical abstraction)

Boolean function of its inputs (e.g., "and," "or," "not"). Using this representational abstraction, circuits containing gates can now be analyzed using Boolean algebra. As the above example illustrates, representational abstraction usually incorporates some measure of functional abstraction. Specifically, we now think of a gate as a Boolean-valued function of its Boolean inputs and analyze the operation of circuits containing gates within the mathematical domain of Boolean algebra, rather than within the electrical engineering domain of voltages and currents.

In programming language terms, we think of each kind of logic gate as an operation within an abstract data type. Each operation ("and", "or", "not", etc.) takes Boolean arguments and has a Boolean result. The operations are abstractions of their respective functions, and the arguments/results are representational abstractions of their respective inputs/outputs.

Encompassing both types of abstraction is the technique of mathematical models. The data of a mathematical model provide a representational abstraction of the modelled object's data, though the components of the model may have no relationship to actual components of the object. The properties of the data visible outside the object, after representational abstraction has been applied, must be identical to the properties of the data external to the model. Note that viewing a flip-flop in terms of logical gates operating on Boolean values is a particular mathematical model in which the components of the model have actual structural components.

Another model of a flip-flop is a flow-table (see Figure III-2b), in which the flip-flop's output q is a function of the inputs and the previous value of q. This function is defined in the table. The representational abstraction is given in terms of Boolean values, as described above. The function (the only component of the model) has no embodiment in the actual components of the model.

Modelling is the most powerful form of abstraction, because the model of an object can be structured independently of any of the object's possible realizations.

The different kinds of abstraction have all been applied to software systems. Functional abstraction is sometimes called "procedure abstraction"; an example of procedure abstraction is a compiler that is composed of a parser and a code generator. Representational abstraction is sometimes called "data abstraction"; an example of data abstraction is the "integer" operations of a hardware machine, which are actually operations on a representation of integers as bit strings. Mathematical modelling corresponds to external specification of a program's behavior; an example of specification abstraction is the use of the external specification (i.e., a mathematical model) of a file system (instead of the code) to describe its behavior. HDM supports all three types of abstraction where applicable.

Different views are possible in forming abstractions of an object, depending on the properties that one wishes to describe or study. For example, the concept of an automobile implied by the owner's manual is different from that implied by the service manual, which is in turn different from the molecular interactions that govern the behavior of the automobile. As discussed earlier, the difference between the object and its abstraction depends on which properties have been abstracted, and on the distance (or grain) or the abstraction.

A basic engineering paradigm is to (1) design a system to meet particular specifications and (2) analyze the system to verify that it does meet its specifications. Both specification techniques and analysis techniques make use of abstraction, and promote understanding and and communication. The difference between software and other engineering disciplines is that in other engineering disciplines the nature of the abstraction is either inherent -- as with a flip-flop composed of electronic components -- or has been discovered as the product of years of study (as with gates). In software the gap between the low-level components (e.g., the hardware or the programming language) and the completed system is great.

Standard sets of intermediate abstractions would be useful. As new problem areas in software are explored, new software abstractions can be expected to emerge.

B. Hierarchies of Abstract Machines

An initial step in applying the concepts of abstraction to software development was the "hierarchy of abstract machines" approach of Dijkstra [2].

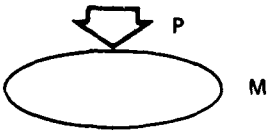
Any programming problem can be posed as follows: "Write a collection of programs P, to execute on an abstract machine M. The programs must have the following properties: ..." (See Figure III-3a.)

For the purposes of HDM, an abstract machine consists of (1) a set of internal data structures, which define its state at a given moment, and (2) a set of operations, by which the internal data structures can be accessed or modified. Each abstract machine has an abstract machine language in which abstract programs (denoting sequences of abstract machine operations) can be written. For some hardware machines, like the IBM 370, the machine language is relatively primitive when compared to the machine language for direct execution machines such as an APL or Pascal machine.

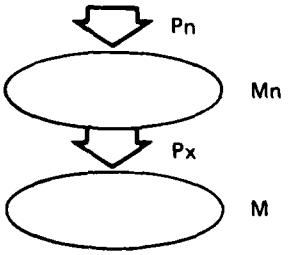
An abstract machine's internal data structures are encapsulated, i.e., accessible only through the operations of that abstract machine. From the vantage point of the abstract machine, its operations may be thought of as indivisible. With indivisible operations, a new operation cannot be invoked until the operation in progress has terminated. Thus, any set of values for the machine's internal data structures must have been reached by a finite sequence of operations (assuming a correct initial state and correct operation). Although indivisibility is not necessary in implementation, it provides a convenient conceptual way of viewing the design.

For purposes of illustration, the set of possible values of the internal data structures of an abstract machine can be viewed as the state space of the abstract machine; a particular assignment of values to the internal data structures can be viewed as a state or point in the state space -- See Figure III-4a. An operation invocation, resulting in a state transition, is represented as a directed arc from one state to another -- Figure III-4b -- and a program invocation, or sequence of

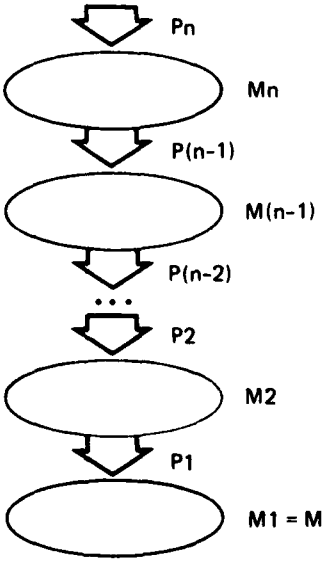
Figure III-3: Hierarchies of Abstract Machines



(a) A SET OF ABSTRACT PROGRAMS P RUNNING ON AN ABSTRACT MACHINE M

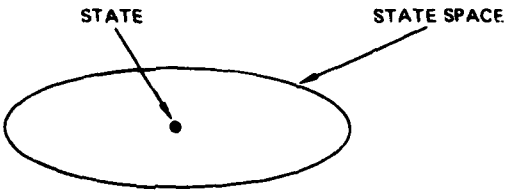


(b) AN ABSTRACT MACHINE Mn REALIZED BY A SET OF ABSTRACT PROGRAMS Px ON ANOTHER MACHINE M

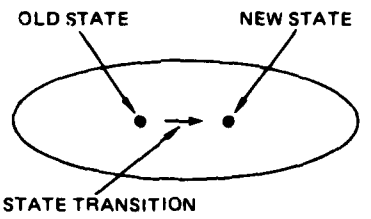


(c) A HIERARCHY OF ABSTRACT MACHINES

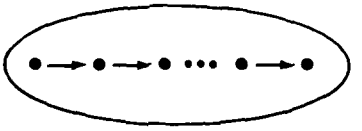
Figure III-4: The State View of a Machine



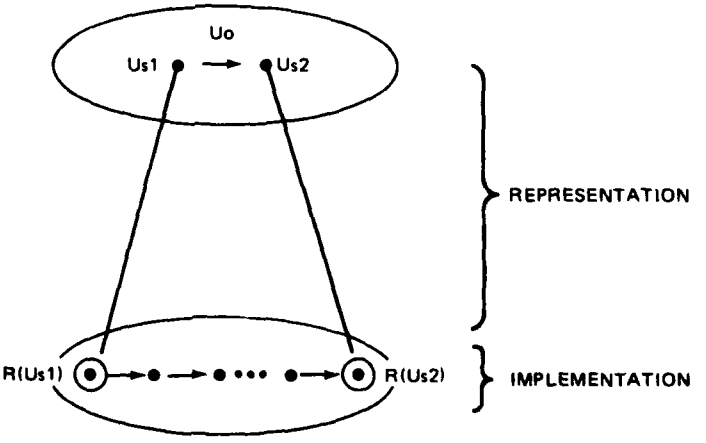
(a) A STATE AND A STATE SPACE



(b) AN OPERATION INVOCATION (A state transition)



(c) A PROGRAM INVOCATION (A sequence of state transitions)



(d) REALIZATION OF AN ABSTRACT MACHINE IN TERMS OF ANOTHER

state transitions, is denoted by a sequence of these arcs -- Figure III-4c.

A complex programming problem is generally difficult to solve by writing the programs P in one step. Instead, another set of programs Pn might be written, whose function is equivalent to that of P, and that (logically) executes on an abstract machine Mn. Executing Pn on Mn must produce the same external results as executing P on M. If Mn is selected properly, then the writing of the program set Pn can be a much simpler task than that originally required for the collection P. Since Mn may not have a physical embodiment, it is necessary to realize Mn in terms of M in order to achieve the ultimate goal of a running system. That is, each operation of Mn is implemented an abstract program running on M. Note that Mn's internal data structures are a data abstraction of those of M, and its operations are a procedure abstraction of a set of programs running on M. Note also that a specification for Mn is a model for any of its realizations.

In the state view of abstract machines, realization can be described as follows. Each operation invocation of the upper machine is implemented by a program invocation on the lower machine. Thus, a new programming problem emerges, that of implementing the operations of Mn as a set of programs running on M (see Figure III-3b).

If Mn is significantly more abstract than M (as is the case in large systems), then a sequence of abstract machines

$$M2, \ldots , M(n-1)$$

may be used to bridge the gap between M and Mn. (For notation's sake, we refer to M as M1.) Given the sequence of abstract machines M1, ... , Mn and program set Pn, we must also construct program sets

$$P1, P2, \ldots , P(n-1)$$

where program set Pi runs on machine Mi and implements the operations of machine M(i+1), for $1 \leq i < n$. Each program in set Pi implements an operation of M(i+1). The target machine M1 is known as the primitive machine. All other abstract machines are non-primitive. A graphical view of a hierarchy of abstract machines is shown in Figure III-3c.

A familiar example of the abstract machine concept is the use of

microprogramming for families of hardware machines. For example, all members of the IBM System/370 family have the same instruction set -- denoting a single abstract machine -- which is realized on different hardware architectures through microprogramming. The physical realization of a 370/168 may be (and is) totally different from that of a 370/135, yet both machines present the same abstract machine to the user.

Similarly, a programming language interpreter presents an abstract machine to its users. Though two LISP interpreters may have completely different implementations and even be running on different machines, the user still sees only a LISP interface, and can execute only LISP commands (i.e., instructions on the LISP abstract machine).

Abstract machines have also been used to describe operating systems, message processing systems, data base systems, and components of compilers and verification systems.

For each two adjacent levels in a hierarchy of abstract machines, each upper-level state maps to a set of lower-level states, and two distinct upper states must map to disjoint sets of lower states. The *implementation of an upper-level operation* relates the upper level to the lower level in the following manner. Let Uo be the implemented upper-level operation, Us1 the initial upper state, and Us2 the final upper state (i.e., Uo takes Us1 to Us2). Furthermore, we let R denote the set-valued representation function that takes an upper-level state as argument and returns the set of lower states used to represent the given upper one. Now, the implementation of Uo on the lower machine will start in some state in R(Us1) and end up in some state in R(Us2). It may go through other lower intermediate states on its way, but just as long as it takes some state Ls1 in R(Us1) to some Ls2 in R(Us2), the implementation is correct. The situation is illustrated in Figure III-4d.

Note that since the upper level may hide some properties of the lower level, it is possible for some lower states to not represent an upper state. Correspondingly, the lower machine may provide some capabilities that are not available to the user of the upper machine.

For example, consider the case of an operating system viewed as an abstract machine running on top of another abstract machine, the machine hardware. The user of the operating system (here, the "upper" machine) will have many capabilities, though it is unlikely that he will be permitted to directly access and modify certain "privileged" information that the bare user of the hardware does have access to.

The hierarchies of abstract machines approach mitigates the problems of producing large software systems in various ways. The abstraction provided by a non-primitive abstract machine hides complexity from the users of that machine. Its definition can be relatively simple even if the implementation is necessarily complex.

C. Modularity

A modular system is one that is divided into easily replaceable parts called modules. To be "easily replaceable," a module must have a well-defined external interface that should be the only information needed to produce a replacement. Thus, a module should be replaceable by any module with the same external interface, even if the new module has a different implementation.

In most engineering discipines, modular systems are commonplace. Attempts to achieve modularity in software systems seem to have often failed -- particularly in large systems -- largely because the interfaces among components were not well understood. For example, a component might have many external interfaces, such as a shared variable or a common data format, that are not explicitly considered to be part of the interface by the system designers. An attempt to replace such a component (or to change it) would often lead to a catastrophic system failure, because some implicit part of the module's interface was not considered. Most software systems today suffer from such "unwritten assumptions."

An attempt to achieve modularity in software systems should have two steps:

* Define what each software module is and what should be

described in its interface.

* Define the interface to each software module.

The first step is described in this section, and the second is described in Section III.D.

Parnas [12] has found that certain forms of modularity are particularly useful in confronting the difficult problems associated with many phases of the software development process. He has attempted to identify specific criteria for decomposing a system into modules, and has described his technique by providing the following characteristics of a module in such a decomposition.

* A module consists of a set of programs that can be invoked by other programs (those outside the module).

* There are two kinds of decisions made in designing the programs of a module: those that define the external behavior of the module, which are available to all programs outside the module; and the remainder of the decisions, which are hidden from use by the programs outside the module. Examples of decisions that should be hidden are data formats, accesses to a common data structure, and specific implementations of a particular program of the module.

* The activity of implementing a module (i.e., writing its programs) can be carried out by one programmer, who may have no knowledge of how other programmers are implementing other modules.

* Two sets of programs can advantageously be made into separate modules if the decisions hidden by each set of programs are independent. This separation is particularly important in decisions that are likely to change, e.g., data formats. One should encapsulate in a module containing no other decisions the programs that have access to the changeable decisions. This will insulate the remainder of the system from the effects of possible changes to these decisions. (Note that the modularity of the implementation need not be identical with the modularity of the abstract machines. For example, several abstract machines may be implemented jointly. Similarly, separately compilable programs may be bound or jointly compiled for execution efficiency.)

These criteria are loosely known under the terms "information hiding" and "module decoupling".

The key problem addressed by modules is that essentially all useful

systems are subject to frequent modification in their lifetime, and that it is important for any technique of structuring to localize the effect of the likely changes.

D. Formal Specification

Achieving the goal of modular systems requires that a programmer understand the impact of invoking a module's operations without knowing its internal details. To portray the behavior of a module independently of its implementation, Parnas has suggested a form for writing module specifications [14]. Parnas' specification technique describes each operation, in a semi-formal way, in terms of changes to the module's (internal) data structures. However, as an ultimate goal he advocates formal specifications.

Although the need to specify a system is generally accepted, the use of informal rather than formal specifications is superficially appealing -- because less effort is generally required to read and write informal specifications. However, informal specifications suffer from the following deficiencies:

* They are often incomplete, with respect to both the desired effects and the anticipted exceptional conditions.
* They can be ambiguous.
* They cannot be checked for consistency.
* They often include (and prematurely bind) details of implementation.
* They cannot be used as the basis of proofs that the (informally) specified system is correctly implemented.

The shortcomings of informal specifications can be overcome by employing a more rigorous specification language whose syntax and semantics are formally defined. An example of a formal specification language is first-order predicate calculus. Specifications written in a formal specification language can be machine-checked syntactically for well-formedness and semantically for self-consistency and consistency with other specifications. (Determination of consistency is in general

an undecideable problem, though many aspects can be checked automatically). Ideally a well conceived specification language should not sacrifice readability, while at the same time be sufficiently rich to characterize completely the effects and the exceptional conditions of each specified operation.

The specification language we have developed for HDM is called SPECIAL (Specification and Assertion Language). One major goal of HDM is to make SPECIAL completely formal. At the time of this writing, the goal has only been partially met. We have developed a formal semantics for a substantial but not complete subset of SPECIAL [1], and work continues on formalizing the entire language.

E. Formal Verification

Testing is the approach most commonly used to decide whether a program, or an entire software system, meets its requirements. However, testing is inadequate because

* It is impossible to test all inputs to many programs of even moderate-size.
* As Dijkstra has stated, testing reveals only the presence, and not the absence, of errors in a program.

An alternative method of analyzing programs is formal verification, from which it is theoretically possible to demonstrate with a rigorous mathematical proof that a program meets its formally stated requirements.

Formal verification is the act of proving mathematically that a program's actual behavior is consistent with some specification of its intended behavior. Many techniques for verification have been developed. Using the inductive assertion technique developed by Floyd [6], the specification is written as a pair of assertions, or predicates in first-order logic, that describe the expected relationship among the values of the variables or data structures before and after the execution of a program. The input assertion places constraints on the values of the input variables, and the output assertion indicates the intended relationship between the values of the output and input

variables. Given a program, a pair of assertions, and a formalization of the semantics of the programming language, it is then possible to construct a theorem (called a verification condition) whose validity implies that the program meets its specifications.

In terms of the state representation shown in Figure III-5a, the assertions represent subsets of the state space. Verification establishes that if the initial state of a program's execution occurs within the subset denoted by the input assertion, then the program will terminate and its final state will fall within the subset denoted by the output assertion.

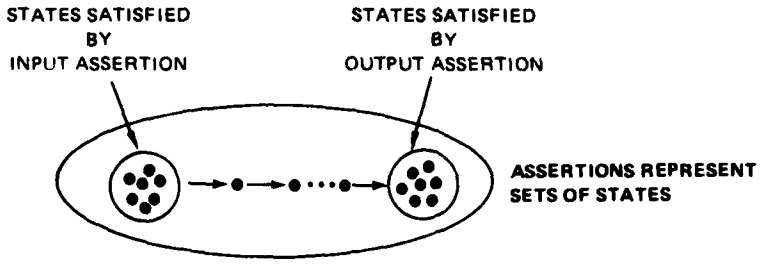
So far, the success of verification has been limited to moderate-sized systems (e.g., 1000 lines of code in a higher-level language) in restricted problem areas. Verification needs much development before it can be applied to large systems. At this time, verification technology is limited by the following problems, as the systems on which it is employed become larger and more applied:

* The assertions are too long and difficult to formulate.
* The programs are too large.
* Most programming languages are semantically too complex to formalize.
* Most application domains are too complex to formalize.
* The relationships among the many programs in the system are often too complex to formalize.
* It is difficult to tell whether a given set of assertions actually states a system's requirements.

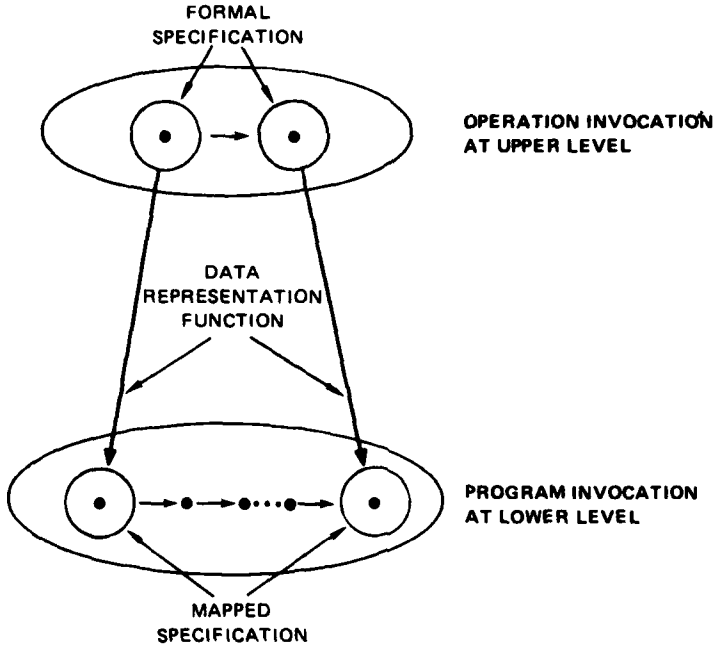
There are several possible ways to improve the program verification process:

* Increase the power of automatic theorem provers to be able to prove theorems resulting from more complex assertions and a richer set of application domains.
* Structure the proof of the program into many small proofs, rather than a few large proofs.
* Restrict the semantics of programming languages and the

Figure III-5: Program Verification and Data Representation



(a) VERIFICATION OF AN ABSTRACT PROGRAM



(b) DATA REPRESENTATION AND HIERARCHICAL VERIFICATION

relationships among programs so that they can be formalized.

Recently it has been recognized ([11], [10], [9]) that even if the verification of a system is not completed, important benefits can be had by designing a system so that it can be verified at some future time. The process of preparing a system for verification leads to a deep examination of all decisions made in system development. As a result of this examination, a better system results.

F. Data Representation

Many applications of programming are based on mathematical concepts (e.g., set theory, numerical mathematics, linear algebra). Often a set of programs implements a mathematical model, meaning that the model is a specification of the set of programs. However, it is not always possible to verify the implementation of the model, because the model may be expressed in different terminology from that of its implementation. For example, operations on sets (e.g., insert an element into a set, delete an element from a set, check if an object is an element of a set, choose a random element from a set) can be implemented as a collection of programs operating on arrays, the contents of an array representing the elements of a set. In set theory, the operations on sets are typically described in terms of the concepts of "S is a null set" and "x is an element of set S." These concepts, however, have no meaning with regard to arrays.

Hoare's proposed solution [8] to this problem requires the definition of an additional relation called a data representation function. This function defines the data of the mathematical model in terms of the data of the implementing programs. A data representation function can be "applied" to the specification of an operation written in terms of a mathematical model, resulting in an assertion (called a mapped specification expressed in terms of the data of the implementing program. Proof of correctness, using the techniques of program verification described above, can then proceed. For the set example, we may represent set S by an array A and an integer variable C (denoting the cardinality of S). An element e is in S if and only if e occurs in

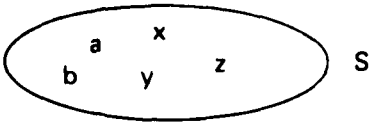
the subarray (A[1] ... A[C]).

See Figure III-6 for an illustration of this representation function for a set with five elements. From this representation function, it can easily be seen that a null set is expressed by any array in which C = 0.

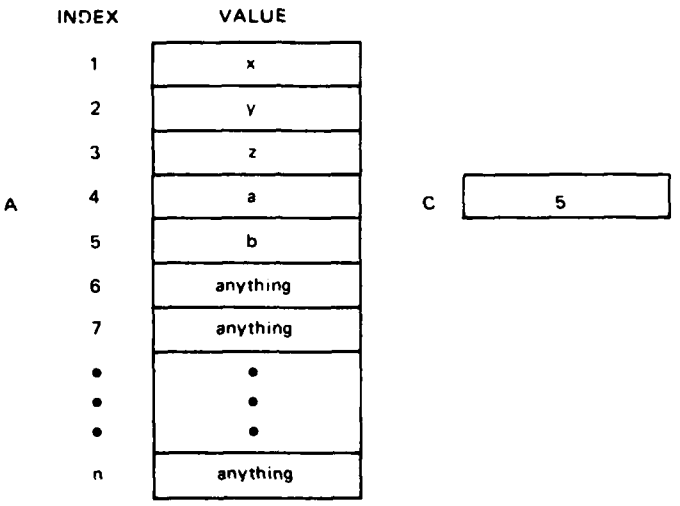
Robinson and Levitt [17] have extended this idea to the verification of hierarchies of formally specified abstract machines, where the mathematical model is the specification of an abstract machine and the set of implementing programs is a set of programs running on a lower-level machine. Note that in this proof technique, the system is structured hierarchically, and the proof is structured in the same way as the system. For a two-level hierarchy, an illustration of data representation in terms of states (Figure III-5b) shows how the states of an operation invocation on the upper-level machine are represented by the states of a program invocation on the lower-level machine. This approach is immediately extendable to an arbitrary number of levels.

In addition to their role in formal verification, data representation functions serve as an important aid in the design process *in general.* First it must be pointed out that the need to represent data is universal in software development (e.g., all different kinds of data in a system are represented in terms of machine words, which are nothing more than bit strings). In most cases these representation decisions are expressed using informal techniques, such as pictures and prose. However, by requiring a formal statement of the data representation function, the designer is obliged to consider certain aspects that might otherwise escape his attention (e.g., which data is hidden, the ordering of data items in a data structure). Such important issues might escape his attention until much later, at which time the implementation code might require significant modification. The use of data representation functions is one example of how a mechanism that was originally developed only for proof has also shown its usefulness in more general aspects of software development (the use of formal specifications is another).

Figure III-6: A Set Represented as an Array



(a) A VALUE OF SET S



(b) THE VALUE OF A, C THAT REPRESENTS S

$$S(A, C) = \{A[i] \mid 1 \leq i \leq C\}$$

(c) A REPRESENTATION FUNCTION FOR S IN TERMS OF A AND C

G. The Decision Model

In developing any software system, decisions are made (either explicitly or implicitly) that in combination ultimately determine all aspects of the system -- its behavior, performance, maintainability, etc. Most views of software concentrate on the system as it exists at a given point in time (the product of those decisions) rather than the decisions themselves (the process by which the system has been developed). Information about decisions can be extremely valuable in software development no matter what development method is used. Parnas [13] was one of the first to recognize the importance of recording decisions, and elevated them to a primary descriptive role in his research on system families.

In the decision model, a software system at any stage of its development results from a sequence of decisions, in which each decision in the sequence is dependent only on the ones occurring before it (see Figure III-7a).

A decision may pertain to any aspect of software development: design (e.g., to build an operating system around a centralized virtual memory mechanism), data representation (e.g., how to represent the queues in the operating system's scheduler), and implementation (e.g., to use a "least-recently-used" algorithm for page replacement). There can be much discussion about exactly what a decision is and how it can be described, but in this document a decision will exist only as a conceptual tool, without recourse to formal definitions or notations.

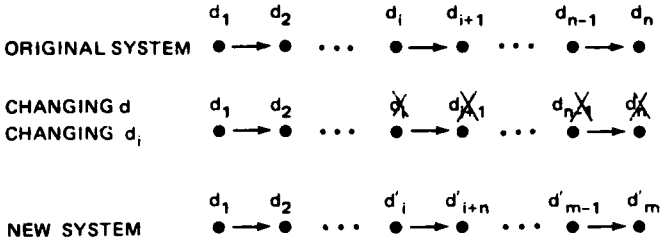
Many difficulties in systems -- with their structure, development, and documentation -- can be examined using the decision model. From several observations about decisions, certain problems in software can be traced, and certain requirements outlined for solutions. The other concepts of HDM can be shown to aid in the management of decisions according to this analysis.

1. The Importance of Early Decisions

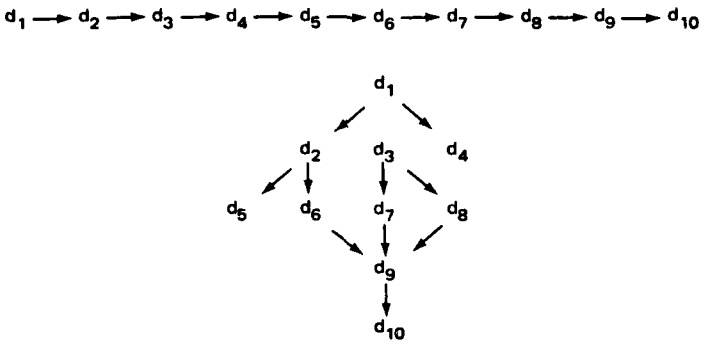
Because a decision is dependent only on certain previously made decisions, the early decisions in software development are those that

Figure III-7: The Decision Model

(a) A SEQUENCE OF DECISIONS



(b) CHANGING A DECISION



(c) INTERDEPENDENCY GRAPH OF DECISIONS IN A SEQUENCE

have the largest number of decisions depending on them. If a decision is changed, potentially all subsequent decisions may have to be changed (see Figure III-7b). Thus, the early decisions are the most important ones, because they potentially require the greatest number of later decisions to be changed. A change in a decision may occur as a result of adapting the system to meet changed requirements, of fine-tuning a system, or of fixing a bug. When an early decision is changed late in the software development process, it is often necessary to redo a system almost completely, even if the change is a small one. Several possible solutions can be proposed:

* Make the decisions first that are least likely to change. Postpone those that can easily change during system development. This is simply a matter of ordering decisions in a particular way, without necessarily changing them. An example is the decision to have a particular data item in the system. The presence of this item may be a basic decision that has to be made, but the format of this item or its location relative to others can be postponed to a later time.

* Make decisions in favor of generality and adaptability. Too often, designers view the system as immutable and static. Design so the system can be modified easily in response to inevitable change. For example, a decision to support any constant maximum number of processes is more general than the decision to support at most 32 of them. It is desirable to parameterize such constants whose values may in fact subsequently have to be changed.

* Minimize the dependency on earlier decisions where possible, and encapsulate related decisions into common modules or abstract machines.

* The concern for "efficiency" is often a false and misplaced one. Many systems suffer from premature optimization that wires-in certain hard-to-modify decisions but does not noticeably improve efficiency. If the system is designed in a modular manner, one can identify from the running system the consumptive modules and replace them with more efficient implementations.

* Evaluate all decisions made before proceeding. Thus, if a bad decision is made and detected almost immediately, there is little penalty incurred in having to redo subsequent decisions. This is especially necessary early in the development process. For example, several levels of evaluation are probably needed before what is usually considered a "design review". When the design review finally

occurs, many decisions have already been made about representation and implementation, without questioning the basic system decisions on which the former are based. This can lead to much redoing of effort if the early decisions are changed.

* Emphasize the process by which the earliest decisions are made. The early design decisions are often specified informally and are traditionally not subjected to as rigorous an analysis as the final code -- which must work properly. However, the early decisions deserve more time and effort than the later ones, if they are going to have the effect of increasing the cost of backtracking.

* Document the system according to the time at which decisions are made. Too often the record of the time ordering of decisions is lost in an unstructured document, or worse yet, in the code. It is often difficult to maintain a system in which a record of individual decisions is not kept. This is also of value durinng backtracking (design changes followed by corresponding implementation changes).

2. The Importance of Decision Interdependence

According to the decision model, for a given decision di in the sequence S = (d1, ..., dn), every subsequent decision dj ($i<j\leq n$) in S may be dependent on di. In actuality, many of the dj are independent of a given di. A more accurate version of the decision model may be constructed as a partial ordering instead of a sequence, illustrating the true dependency among decisions. (In the example of Figure III-7c, no decisions are dependent on d4, while several are dependent on d7.) This partial ordering may also be applied with clusters of decisions at a node, instead of a single decision. Difficulties in system development and maintenance can occur because decisions in a system are too interdependent or because the interdependence among the decisions in a system is not precisely known. Two solutions to these problems are as follows:

* Reflect the interdependence of decisions in the system documentation.

* Structure a system so as to minimize the interdependence among groups of decisions.

3. The Profusion of Decisions

Many of the problems that occur in managing software development decisions would not be serious if there were only a small number of decisions. However, even in a small software system, there are hundreds -- maybe thousands -- of decisions to be made. (See the example of the use of HDM in Volume III of this handbook.) Many difficulties in software result from this profusion of decisions.

In such cases, it is difficult to keep track of which decisions have actually been made. Thus, a common error in software development is to recognize the need for a decision that has in fact already been made -- either earlier by the same person or in some other part of the system by someone else. When these two (identical) decisions are resolved consistently, there is no problem (at least initially); however, if they are made differently, the system is unsound. For example, an argument to a procedure may be informally assumed to be a non-negative integer; the code may handle only a positive integer, blowing up when it is given zero as an argument.

Two guidelines are useful in managing the proliferation of decisions.

* Write down all decisions. Obviously, this is easier said than done. Unless there is a framework for what constitutes a complete set of decisions, this will be extremely difficult.

* Write decisions precisely. Mathematical English (the language of mathematics textbooks) can be used, but is difficult to check for syntactic and semantic consistency. Some formal language is probably the answer.

4. The Scattering of Decisions

Even if the decisions of a system are to be written down, conventional methods allow the recording of the same decision to be scattered throughout the system. An example of decision scattering is a shared data format. As a result, it may seem that a decision is made in only one place, or a few places, when in actuality it is made in many other places that are difficult to track down, especially in a large system. When this decision is changed, but not every embodiment of it

is changed, the result is an inconsistent system. Two aids to combat decision scattering problems are as follows.

* Structure the system so that a single decision is localized as much as possible to a small part of the system. This minimizes the scattering phenomenon.

* Make explicit all sharing of decisions among parts of the system. Thus, when a decision is scattered among parts of the system, the scope of the scattering is well-defined.

5. Relation of Decisions to Other HDM Concepts

Many of the concepts of HDM described above (e.g., abstraction, hierarchical structure, modularity, formal specification, formal verification, data representation) are closely related to the decision model. Abstraction, hierarchical structure, and modularity provide means for structuring and grouping large numbers of decisions in particular ways. This grouping occurs both according to the time when decisions are made (by separating the development activity into stages -- e.g., definition, representation, and implementation of abstract machines), and according to dependency (by categorizing groups of decisions within modules and levels). Levels of abstraction and modules provide a way of describing the interdependence of decisions and criteria for minimizing that interdependence. Modules are also a way of localizing decisions to a single part of the system. Formal specifications at all stages of development provide a means of documenting all decisions in a precise and complete manner. Formal verification provides a means for deciding the consistency among decisions. The decision model is a useful vehicle for formulating may issues in software development.

Subsequent discussions of HDM in this handbook make frequent use of the decision model, either for justifying a particular feature of HDM or for discussing particular issues in developing actual systems using HDM.

H. The Contributions of HDM

The concepts of Dijkstra, Parnas, Floyd, and Hoare are significant advances in understanding the complexity of software and the difficulty

in developing large software systems. The development of these concepts has been accompanied by great expectations, i.e., that their use would cause a significant improvement in the quality of software and a reduction in its total cost. Previous attempts to use such concepts have fallen far short of expectations, because each concept in isolation is not powerful enough to solve the entire software problem and because the concepts have no languages and tools to support them.

The major contribution of HDM is to embody these concepts into an integrated approach that is supported by languages and tools. Some of the concepts such as abstraction and modularity are reflected in particular components of HDM (e.g., abstract machines and modules), while others, such as formal specification and the decision model are reflected throughout HDM. Specifically, HDM is derived from these concepts as follows.

* The "hierarchy of machines" approach of Dijkstra provides the basis for HDM. The "data representation" method proposed by Hoare, and by Robinson and Levitt is also incorporated.

* The "module" concept of Parnas is used to define units of decomposition for abstract machines, data representations and implementations.

* Formal specifications are used to specify all of the components of a system (e.g., abstract programs and abstract machines) and their hierarchical interconnections. Remember, though, that at the present time complete formality is more of a goal than a reality.

IV THE BASIS OF HDM

This chapter shows how the foregoing concepts are integrated into HDM, including the use of modules, abstract machines, abstract programs, data representations, and implementations.

A. Hierarchical Structure in HDM

In HDM, a software system is viewed as a hierarchy of abstract machines. Each abstract machine can be viewed as providing a particular set of capabilities or facilities through a particular interface to its user(s) (e.g., an operating system, a data base manager, or an interpretive language environment).

Each abstract machine is first specified as a distinct, independent entity. Next (in the data representation stage), its data structures are defined in terms of the next lower level abstract machine. Finally (in the implementation stage), each abstract machine (except for the lowest-level machine) is then given an implementation in terms of abstract programs written in the language of the next lower level abstract machine.

This section provides a detailed description of the basis of HDM sufficient to allow a new user of HDM to begin to think about systems in terms of abstract machines and abstract programs.

1. Abstract Machines and Programs

As mentioned in Section III.B, an abstract machine supported by HDM consists of a set of internal data structures and a set of operations that access or modify the values of the internal data structures. Each abstract machine has the following properties.

* It accomplishes work by executing sequences of operations on behalf of its users (e.g, a program or a human). It provides a self-contained environment, having neither side effects nor additional operations.

* It receives input either through arguments to its operations or by fetching values from those of its internal data

structures that represent input devices.

* It produces output either as the result of one of its operations or by storing values in those of its internal data structures that represent output devices.

* It encapsulates its internal data structures, i.e., does not allow access to or modification of the data except through the operations of the abstract machine. The effects of its operations are specified only in terms of the abstract machine's own internal data structures.

* It provides operations that appear to be indivisible. This means that the internal data structures always appear externally to be in a self-consistent state. (Note that this has interesting implications for the specification of internal parallelism.)

* It can operate in an environment containing asynchronous events. In such an environment, multiple users or asynchronous programs can share the use of an abstract machine, or asynchronous programs can be used to implement its operations. As in a sequential (synchronous) environment, an abstract machine running in an asynchronous, parallel environment provides a set of indivisible operations to users and programs. (Asynchronous operation of abstract machines is not a subject of this handbook. Consult [10] for more information.)

An abstract program is a string in a particular programming language which, when interpreted, describes a sequence of operations on an abstract machine. An abstract program has the following properties.

* It contains no permanent data structures of its own. Local variables may be used, but do not survive a particular invocation of an abstract program. All permanent data is stored in the internal data structures of the abstract machine, which must provide whatever data structures are needed (e.g., arrays, strings, lists).

* It may contain calls to subroutines, if provided by the abstract programming language.

* It contains only control statements, assignment statements (if local variables are used), subroutine calls, and invocations to the operations of an abstract machine.

* It may invoke the operations of only one abstract machine. Any subroutines called by the abstract program are subject to the same restriction.

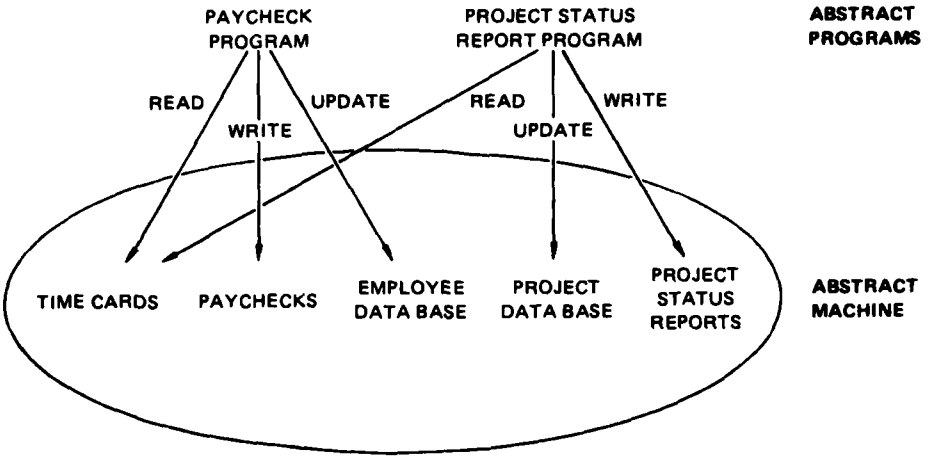
* It communicates with the abstract machine on which it executes by means of the arguments to and results of operation invocations, and by notification of exceptional conditions.

An example of a set of abstract programs running on an abstract machine can be found in a company's management information system, in which employees can charge their working hours to any one of several project accounts. From the employee time cards and from information on projects and employees, programs must generate at regular intervals a paycheck for each employee and a financial status report for each project, while updating the permanent information on both projects and employees. A possible abstract machine to solve this problem would contain facilities representing a time card input file, a paycheck output file, an employee data base, and a project data base. Two abstract programs can be provided, one to produce paychecks -- updating the employee data base -- and one to produce project status reports -- updating the project data base. The system structure is shown in Figure IV-1. This structure becomes more complex as new functions are specified, such as the ability to update the data bases to provide for new employees, new projects, salary changes, etc. Note that the primitive abstract machine provided -- perhaps a hardware machine, an operating system, or a COBOL processor -- is not the machine designed here. However, the primitive machine could be any of these and the same system description would apply. This is an example of the use of abstraction, the postponement of particular decisions -- in this case the nature of the underlying machine -- until later in the software-development process. Note also that the physical storage of the various data entities -- whether on disk, on tape, or in main memory -- is not specified. Again this is an example of abstraction.

2. Abstract Machine Realization

As discussed above, the abstract machine provided in most programming problems is not the one most desired. To develop the desired system, there is a choice between two alternatives: to write extremely complex programs on the abstract machine provided, or to write simple programs on the abstract machine desired and to realize the

Figure IV-1: Abstract Machines in a Management Information System



abstract machine desired in terms of the one provided. There are two aspects to realizing a given abstract machine in terms of the next lower-level machine: (1) the representation of each data structures of the upper machine in terms of the lower-level machine, and (2) the implementation of each operation of the upper machine as an abstract program running on the lower machine.

The step of recording data representation decisions is a very important one, even though it was originally introduced only to construct proofs. When the upper machine is realized in terms of the lower, the data structures of the upper are usually different from those of the lower. The programs that implement the upper machine are thus operating not on the data of the upper machine's specification (i.e., the data structures of the upper level), but rather on the data of the lower machine (i.e., the data structures of the lower level).

The recording of data representation decisions is a vital step towards implementing an upper machine in terms of a lower machine. Essentially, data representations map the state space of the upper level to the state space of the lower. Before actual implementation programs can be written -- programs that manipulate lower-level data structures -- we must know how the data structures of the upper level are to be represented. The relationship between the two sets of data structures is thus a set of decisions made in order to write these implementation programs. If these decisions are not written down, people working on the implementation programs may become confused. In addition, maintenance may become a real problem, because the representation decisions will be "lost" and will have to be reconstructed by reading the code. Data representation specifications are analogous to the diagrams (e.g., control block formats in operating systems) that are often used in conventional methods to describe the same kinds of decision. When the implementation of the abstract machine is finally written, the data representation becomes an invaluable point of reference to both developers and maintainers.

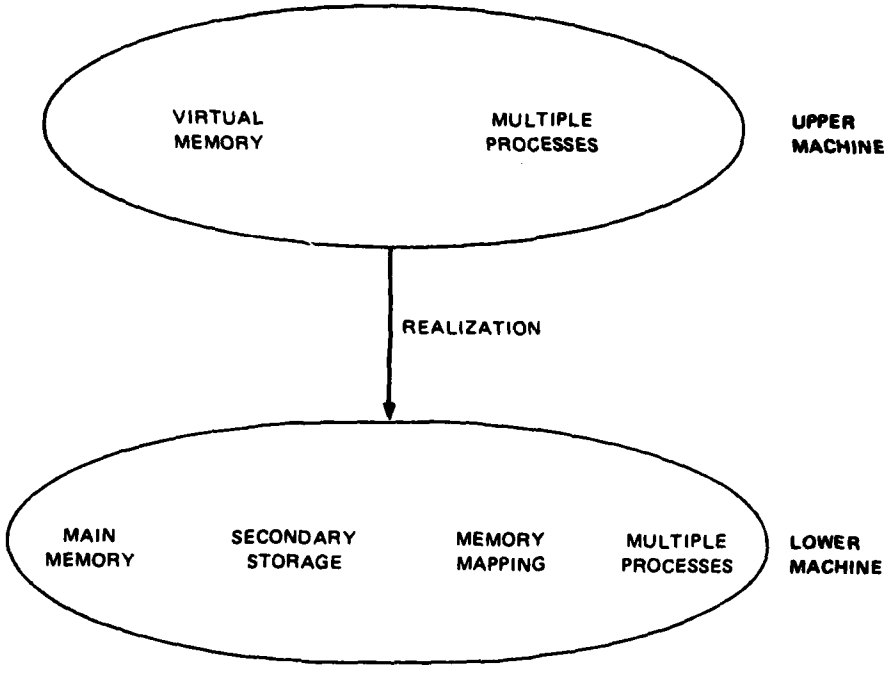
An example of abstract machine realization occurs in an operating system based on a virtual memory ([10], [3], [15]). In this system one

level (or abstract machine) provides a virtual memory, as well as some other facilities (such as multiple processes and process synchronization). The goal of a virtual memory is to allow a program to have a large address space (called a virtual address space), which is potentially larger than the main memory that would normally be available to it. The program would access this virtual address space uniformly, as if it were referencing main memory. This is the abstraction. However, the implementation is different: the information in a program's virtual address space is scattererd throughout primary memory and secondary storage. If the program tries to access a virtual address that is currently resident in main memory, as determined by a lower-level facility called a memory mapping, the memory mapping will provide the proper physical address and the location will be accessed. If, on the other hand, the program tries to access a virtual address that the memory mapping determines is resident in secondary storage, the information will be brought into main memory (using the secondary storage address contained in the memory mapping), the memory mapping will be updated to reflect the new position of the information, and finally the information will be accessed as if it had already been in main memory. The response to the calling program is the same in either case -- the correct information is accessed; in the latter case it just takes longer. The memory mapping maps segments of virtual addresses to physical storage. The ordering of the segments in virtual storage is independent of their ordering in physical storage. The data representation for the virtual memory mechanism could be stated informally as follows:

> If the memory mapping determines that a is represented in main memory, then the contents of virtual address a are the contents of the main memory address mapped to by a; otherwise the contents of virtual address a are the contents of the secondary storage address mapped to by a.

An abstract machine structure for this realization is described in Figure IV-2. The top level contains facilities for the virtual memory and multiple processes -- both to be provided to upper levels in the operating system -- and the bottom level provides facilities for main memory, secondary storage, memory mapping, and multiple processes.

Figure IV-2: Abstract Machine Realization for a Virtual Memory



(Note how the same facility, multiple processes, appears at two adjacent levels. This phenomenon will be an important motivating factor in developing the mechanisms to support modularity.) Note that the top-level machine, containing the virtual memory, is conceptually more simple than the lower-level machine, though the realization of the virtual memory is extremely complex. This illustrates a major goal of HDM: to make systems appear conceptually simple at higher levels in the face of (1) complex lower-level facilities and (2) a complex realization.

3. Hierarchies of Abstract Machines

In many cases the realization of an abstract machine is in itself a difficult programming problem, and cannot be easily accomplished in one step. Thus, in realizing a particular abstract machine Mn in terms of a primtive machine M1, it may be necessary to hypothesize a sequence of abstract machines

$$Mn, M(n-1), M(n-2), \ldots , M1$$

where in each case, Mi can be realized in terms of M(i-1) in a relatively straightforward manner.

A sequence of machines determined to be a solution to a given programming problem is said to be a hierarchical machine decomposition of that problem. Issues raised during hierarchical decomposition lead to a better understanding of the problem to be solved. In the past uses of HDM, the hierarchical decomposition has generally been a highly creative iterative process, although it tends to converge quite rapidly toward a stable framework upon which minor modifications can be made. It is recommended that the writing of complete specifications be deferred until the hierachical decomposition has stabilized, and that the writing of code be deferred until the specifications have been thoroughly checked. There is usually much discussion about a proposed hierarchical decomposition (leading to changes) before a satisfactory one is agreed upon. At this point, the writing of specifications can begin. After all abstract machines are specified, each non-primitive abstract machine is realized in terms of the one below it in the

hierarchy.

One question that frequently arises during the process of hierarchical decomposition is, "What issues determine the number of abstract machines in a hierarchical decomposition?" Several general observations are relevant:

* It is completely the decision of the designer how many abstract machines to have in a hierarchy. The statement "Mi has a straightforward realization in terms of Mj" is a subjective one. Another designer may not agree, and would insert an abstract machine Mk (or maybe more) between Mi and Mj so that he could make the statement.

* The hierarchy may be expanded as the system is developed. The difficulty of realizing a particular abstract machine Mi in terms of Mj may not be fully appreciated when the hierarchy is first conceived. At some later time the designer may choose to add an intermediate machine Mk to simplify the overall programming task. Typically a hierarchy that is adequate for presenting a system may be too coarse for its ultimate realization -- thus the need to add levels.

More exact statements concerning criteria for hierarchical decomposition -- especially what constitutes a straightforward realization -- will be given in Chapter VI.

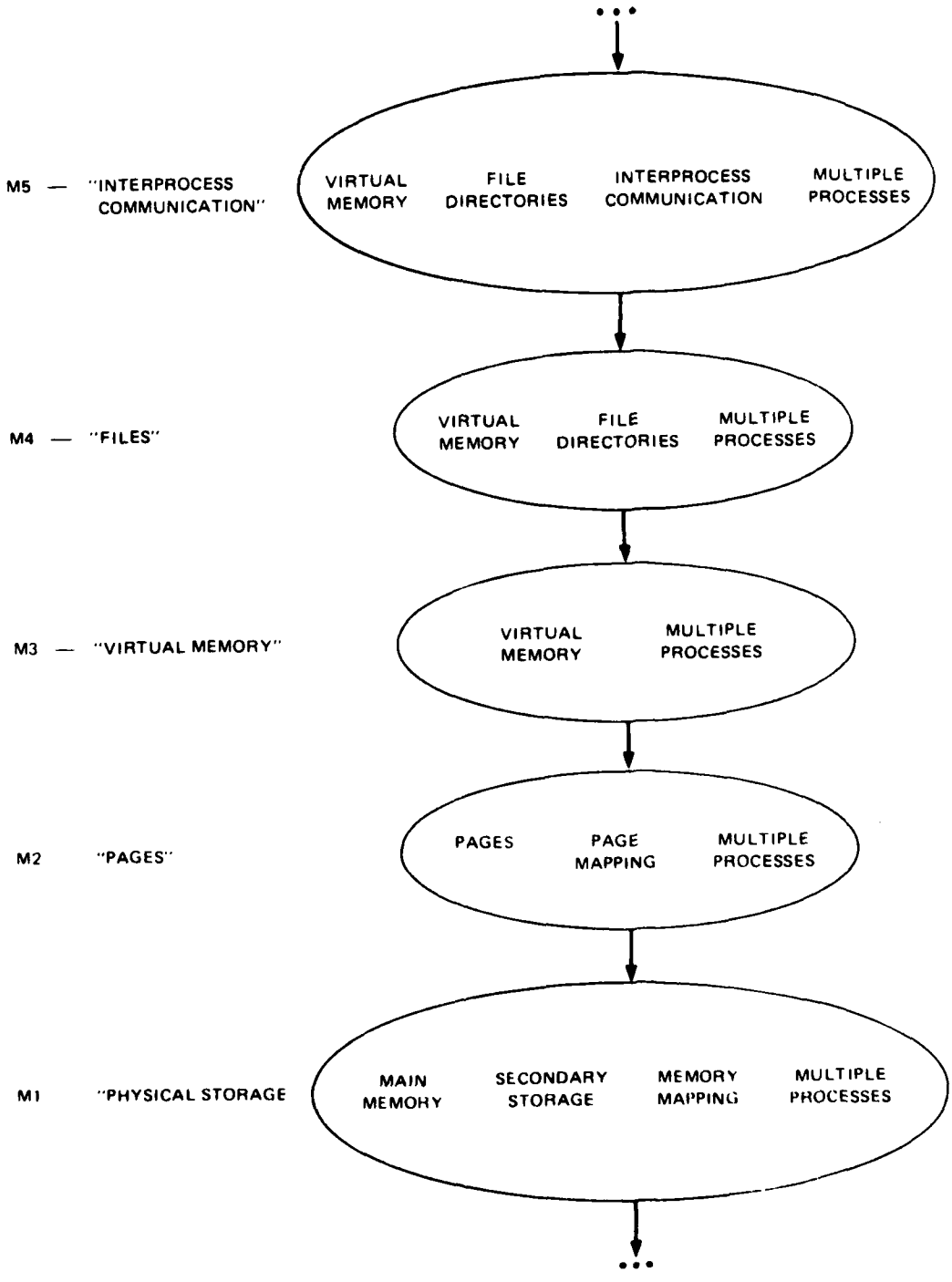
An example of a hierarchical decomposition can be found by extending the operating system example from Section IV.A.2. In this hierarchy, described in Figure IV-3, several levels have been added to the system shown in Figure IV-2: one supporting pages (M2), between the "virtual memory" level (M3) and the "physical memory" level (M1); one supporting file directories (M4), where "files" are segments from the virtual memory mechanism; and one supporting interprocess communication (M5). This partial hierarchy is similar to that of PSOS (Provably Secure Operating System) [10], an operating system that has been specified using HDM.

B. Modular Units of Specification

1. Introduction

The abstract machine model of Section IV.A is sufficient by itself

Figure IV-3: Part of an HDM Hierarchical Decomposition



to describe any software system. However, systems that are structured using the abstract machine model alone do not necessarily reflect Parnas' criteria for modularity [12]. For this reason HDM combines the advantages of the abstract machine model and of modularity in the Parnas sense by requiring the specification of abstract machines, data representations, and abstract implementations to be done in terms of modular units. These units are modules (for specifications), representation clusters (for representations), and implementation clusters (for implementations). These modular units allow a closely related set of decisions to be grouped into a common unit of specification and allow loosely coupled groups of decisions to be expressed in different specifications.

2. Modules

As illustrated in Section IV.A, an abstract machine may consist of different modules. The following attributes of a module are noted.

1. A module is part of an abstract machine. An abstract machine may be described in terms of its component modules.

2. A module may appear in multiple abstract machines in the same hierarchy. The same specification may be used to describe all of the module's appearances.

3. A module's behavior is largely independent of that of other modules within the same abstract machine, but some interconnections among modules in the same machine are permitted, by allowing the specification of one module to reference the entities of another.

4. A module specification (like an abstract machine specification) is written independently of any possible realization. The implementation of a module must be independent of the realizations of all other modules in the system, even those within the same abstract machine. The realization of a module A whose specification references the specification of a module B must depend only on the specification -- and not the realization -- of module B. In order to achieve this attribute, a module's internal data structures must be encapsulated, and its operations must be indivisible (as with an abstract machine).

Attributes (1) and (2) are intuitive. Attribute (3) raises the question, "Why allow interconnections among modules, if modules are

supposed to be independent?" If total independence were allowed, it would be very difficult to decompose certain systems into modules. In actual systems, all kinds of minor decisions are shared, even among modules whose operation is (and whose implementations are) separable. For example, in the operating system example in Section IV.A.3, two abstractions, "pages" and "page mapping," may know about the page size. Apart from that, these two facilities could be specified separately. Attribute (4) guarantees that the interconnections among module specifications permitted by attribute (3) maintain the most important property of modules, complete independence of the realizations of different modules. With these attributes in mind, a module can now be defined, in a manner analogous to an abstract machine.

More formally, a module may be thought of as a triple <O, I, E>, where O is a set of operations, I is a set of internal data structures, E is a set of external data structures, and I and E are disjoint. Each operation in I is defined as a relation that constrains the post-invocation values of data structures in I and E based on their pre-invocation values and the operation's actual arguments. This definition implies that all interconnections among modules are expressed by allowing the operations of one module to access and change the values of the data structures of another. If any of the external data structures of a module A are also internal data structures of a module B, the relation A **ref** B (or "A references B") is true. If

$$C1, C2, \ldots, Cn$$

is a sequence of modules such that A = C1, B = Cn, and

$$Ci \textbf{ ref } C(i+1) \text{ for } 1 \leq i < n$$

is true, then the relation A **ref+** B is true. **ref+** is the positive transitive closure of **ref**. There are two rules concerning the **ref** and **ref+** relations:

1. The set of modules of any abstract machine is closed under the **ref** relation, i.e., if A **ref** B holds, any abstract machine that contains A also contains B. This rule must be enforced to allow the definition of an abstract machine as a complete environment for an abstract program.

2. **ref** must be non-symmetric, i.e., if A **ref** B is true, then

B **ref** A must be false. In terms of **ref+**, this means that **ref+** must not contain any circularities, i.e., A **ref+** A must never be true. When we refer to the non-circularity of the **ref** relation, we are in fact talking about the non-circularity of the induced **ref+** relation. The non-circularity rule must be enforced to guarantee (a) the encapsulation of the module's internal data structures, (b) the indivisibility of its operations, and (c) the independence of the realization of any module from the implementation of any other module. The reason for restriction (c) is presented in Section IV.B.4.

Graphic views of both correct and incorrect uses of both relations are illustrated in Figures IV-4 and IV-5, respectively, where A, B, and C are modules and a directed arrow from A to B means A **ref** B. (Note that **ref** is not a partial ordering, as it is not transitive.)

Thus, an abstract machine can be described as a set of modules. The following observations apply to the relationship between modules and abstract machines:

1. A nonempty set of modules S within an abstract machine M that is closed under **ref** (and thus **ref+**) is said to be a submachine of M. This is because S by itself could define an abstract machine, since its description is self-contained. Note that if S1 and S2 are submachines, S1 UNION S2 is also a submachine. (We use UNION, INTER, and DIFF to denote set union, set intersection, and set difference, respectively.)

2. Two or more modules may be combined to form a single module. Thus, if <O1, I1, E1> and <O2, I2, E2> are modules, then <O, I, E> is their combination, where

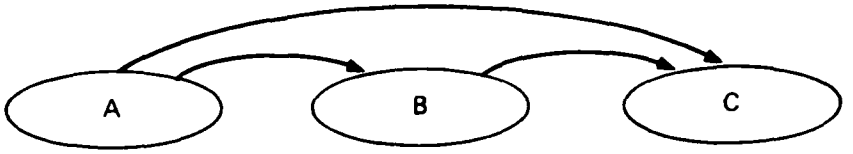
   O = O1 UNION O2,
   I = I1 UNION I2, and
   E = (E1 UNION E2) DIFF (I1 UNION I2).

3. A single module may be decomposed into two or more modules, subject to the non-symmetry of **ref**. Thus, the module <O, I, E> may be decomposed into two modules m1 = <O1, I1, E1> and m2 = <O2, I2, E2>, where

   (O1 INTER O2) = {},
   (I1 INTER I1) = {},
   O = O1 UNION O2,
   I = I1 UNION I2,
   E = E2 UNION E2 and
   m1 **ref** m2 and m2 **ref** m1 are not both true

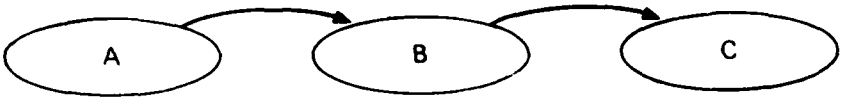
   The criteria for decomposing modules, aside from these rules, are subjective, and are discussed in Chapter VI, as well as in [12].

Figure IV-4: Correct Uses of **ref** and **ref+**



A ref B
A ref C

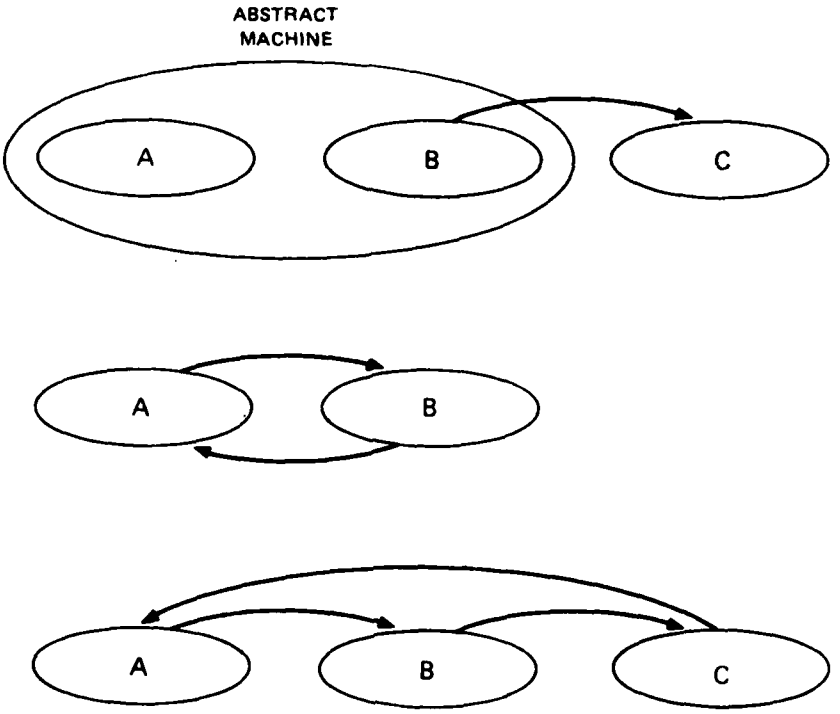
A ref+ B
A ref+ C



A ref B
B ref C

A ref+ B
A ref+ C
B ref+ C

Figure IV-5: Incorrect Uses of **ref** and **ref+**



A module is thus a structuring of decisions in several senses. In terms of the decisions about the external behavior of modules, if A **ref** B holds, then the decisions about the behavior of A are dependent on those about the behavior B, but not vice versa. If A and B are independent (i.e., neither A **ref** B nor B **ref** A holds) then the decisions about the behavior of A are independent of those about the behavior of B. For any two modules A and B, the decisions about the implementation of A are independent of those about the implementation of B.

An example of the **ref** relation can be found in operations that transfer information between an I/O device and main memory. The data structures for the device and for main memory can be put in different modules, "device" and "main_mem". The operation "read," which transfers some information from the device to somewhere in main memory, can be in either "device" or "main_mem". The definition of the **ref** relation will depend upon which module the "read" operation is placed in, as shown in Figure IV-6. (Other operations besides the "read" will be necessary to implement both modules successfully.)

The **ref** relation, however, is not specified directly in module specifications. Rather, it is derived from the **externalref** relation. When module A references either the internal data structures or the operations of module B in a specification, then A **externalref** B holds. Note that if A **externalref** B and B **externalref** C, then it is possible to have A **ref** C without having A **externalref** C. This is because an operation of A may reference an operation of B, which in turn references the internal data structures of C. For an example of the **externalref** relation, see Figure IV-7. Like **ref**, **externalref** must not admit circularities. Because module specifications are written using the **externalref** relation, only this relation (and **externalref+**) will be used in subsequent discussions.

Typically the decomposition of a system into modules occurs concurrently with its decomposition into abstract machines. Each abstract machine of the system "introduces" one or more modules -- those that occur at this level and above but not at lower levels -- and "hides" one or more other modules -- those that occur at lower levels

Figure IV-6: Two Different Modular Decompositions of a Problem

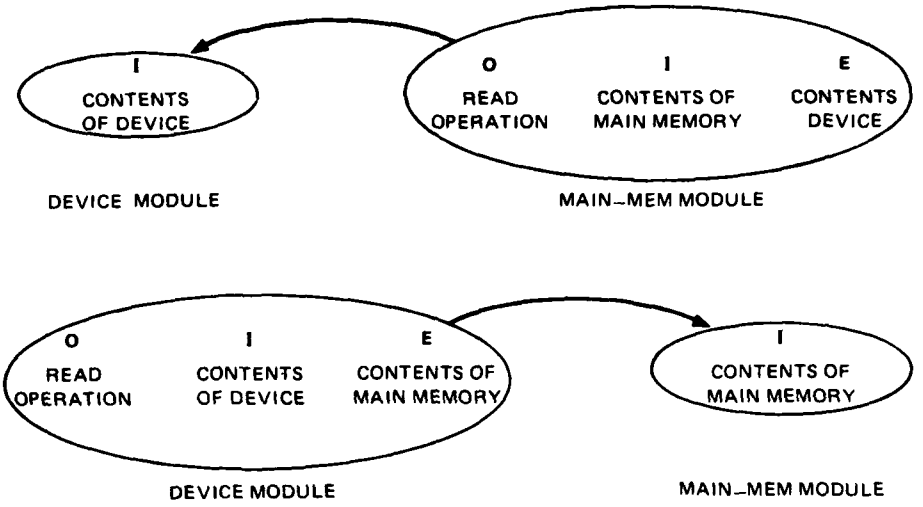
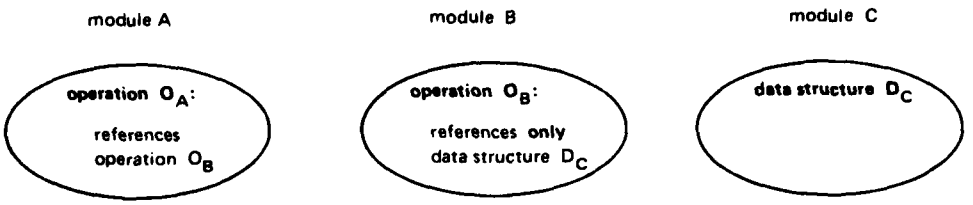


Figure IV-7: The **externalref** Relation



A **externalref** B
B **externalref** C
A **ref** C
NOT A **ref** B
NOT A **externalref** C
A **ref+** C
B **ref+** C
A **externalref+** B
A **externalref+** C
B **externalref+** C

but not at this level or above. For example, in Figure IV-3, where each abstraction can be viewed as having a separate module, abstract machine M3 introduces the "virtual memory" module, while hiding the "pages" and "page mapping" modules.

3. Representation Clusters

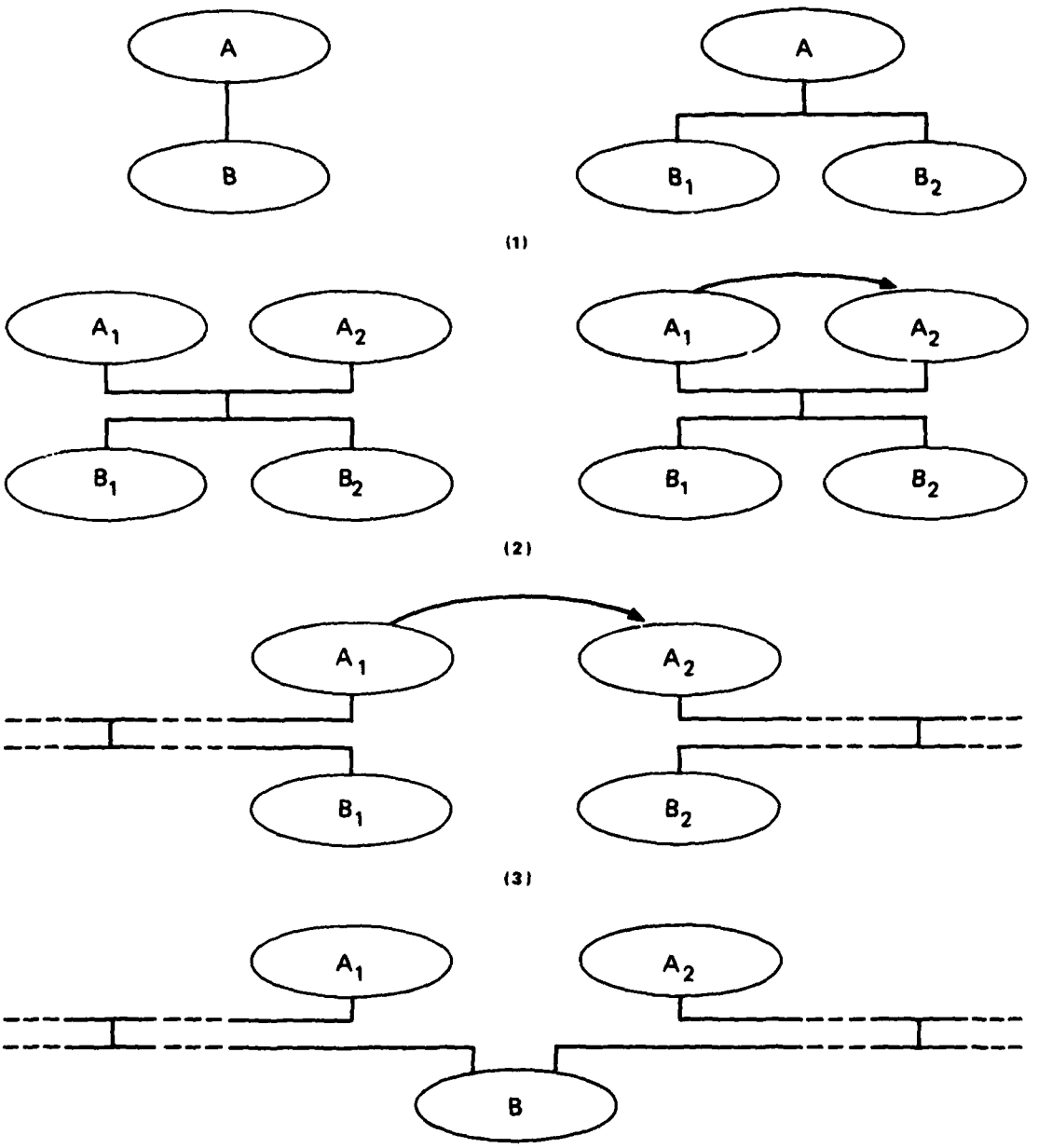
There are several approaches to defining a modular unit for representation decisions. One extreme would be to require a single grouping of representation decisions for an entire abstract machine. Such a grouping is too coarse, because it does not allow the possibility of grouping representation decisions according to individual modules within the abstract machine. The other extreme would be to group the representation decisions on the basis of a single module. Such a grouping is too fine, because it does not allow representation decisions for several upper-level modules to be shared, or grouped together into a common unit. The solution is to allow the grouping of representations into units called representation clusters, which describe the representation decisions for one or more upper-level modules. In this way the software designer can decide whether upper-level modules should be in different representation clusters, indicating a separation of representation decisions, or whether they should be in the same implementation cluster, indicating a sharing of representation decisions.

A representation cluster defines the internal data structures of the upper modules of the cluster in terms of the internal data structures of the lower modules of the cluster.

In some cases a module of the upper machine is not an upper module of the cluster, but is externally referenced by one or more of the upper modules. Though this situation seems anomalous, in fact it is not. An upper-level entity from some other representation cluster may be referenced even though it does not play a part in the given representation. A more complete discussion of this situation is given in the chapter on representations in Volume II.

Four schemas of representation clusters can be described, with each

Figure IV-8: Schemas of Representation Clusters



schema describing different types of decision dependencies. The schemas are illustrated in Figure IV-8.

1. A single upper module (defining a submachine), and one or more lower modules. This represents a maximum separation of representation decisions. The implementation of the upper module is dependent only on the specification of the upper module, on the specifications of all lower modules, and on the representation cluster.

2. Two or more upper modules (defining a submachine), with one or more lower modules. In this case the representation decisions are shared among the upper modules. For each upper module Ai of the cluster, its implementation is dependent only on the specifications of Ai, on the specifications of all upper modules Aj such that Ai **ref** Aj is true, on the specifications of all lower modules, and on the representation cluster. Note that in order for the implementations of all modules in such a representation cluster to be independent, the internal data structures of different modules must map to disjoint sets of data structures in the lower modules.

3. Two modules A1 and A2, where A1 **ref** A2 is true, are upper modules of different representation clusters. In this case there may be some representation decisions (those of certain data types) shared among the representations of A1 and A2. For more details of this phenomenon, please consult Volume III of this handbook, where examples are presented. In any case, the shared decisions are presented redundantly in both the representation clusters of A1 and A2, so that the dependencies are as stated in schemas (1) and (2).

4. A single module is the lower module of two or more representation clusters. For this to be allowed, the condition of schema (2) must be met.

A set of representation clusters C1, C2, ..., Cn define an abstract machine representation of abstract machine M2 in terms of abstract machine M1 if and only if

1. Each module of M2 is an upper module of exactly one Ci ($1 \leq i \leq n$).

2. For each Ci, each upper module is in M2.

3. Each module of M1 is a lower module in at least one Ci.

4. For each Ci, each lower module is in M1.

Note that the same module may appear as both an upper and a lower

module in a given representation cluster, because the same module may appear in different abstract machines in a given hierarchy. In almost all cases, these separate appearances of the same module have the same implementation of their operations. However, these appearances may not always have the same data structures. The following possibilities for data representations arise for when the same module appears in two adjacent abstract machines in a hierarchy, described graphically in Figure IV-9.

1. A simple identity representation of data structures. This is marked with the letters "ID."

2. One or more additional upper modules A1, A2, ..., An "sharing" a representation with a module B. In most cases all data structures in the upper appearance of B are represented identically in B's lower appearance. If this is the case, the data structures of the upper appearance of B are a subset of those of its lower appearance.

3. Sometimes one or more of the "new" modules will reference B. This is permissible, but the implementor of such an upper module must be careful not to confuse the data structures of B that are referenced from those that are representing the data structures of the implemented module.

Of course a module may have a different implementation (with a non-identical representation) in each of two adjacent levels in which it appears. In this case, the two appearances can be different modules when appearing in representation clusters.

An example of an actual representation cluster may contain "arrays" (a module that manages arrays) on the bottom and "stacks" (a module that manages stacks), "queues" (a module that manages queues), and "arrays" on the top. Each stack or queue is represented by a single array and each top-level array is represented identically. Thus, the three upper modules partition the data structures of the lower module, as shown in Figure IV-10a. Note that if the stacks and queues were stacks and queues of arrays, then the relations "stacks" ref "arrays" and "queues" ref "arrays" are true, as shown in Figure IV-10b.

Figure IV-9: Representing the Same Module at Different Levels

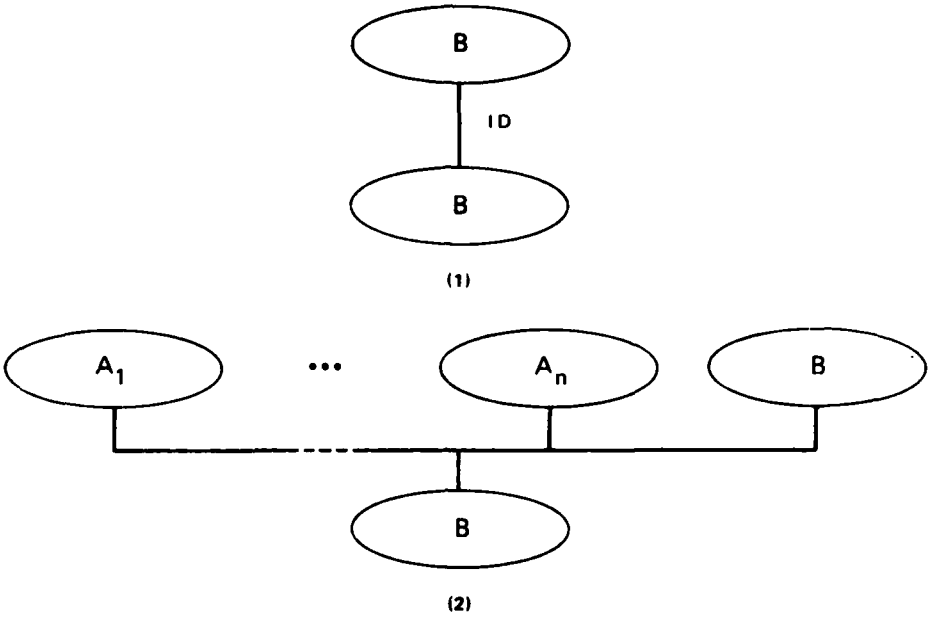
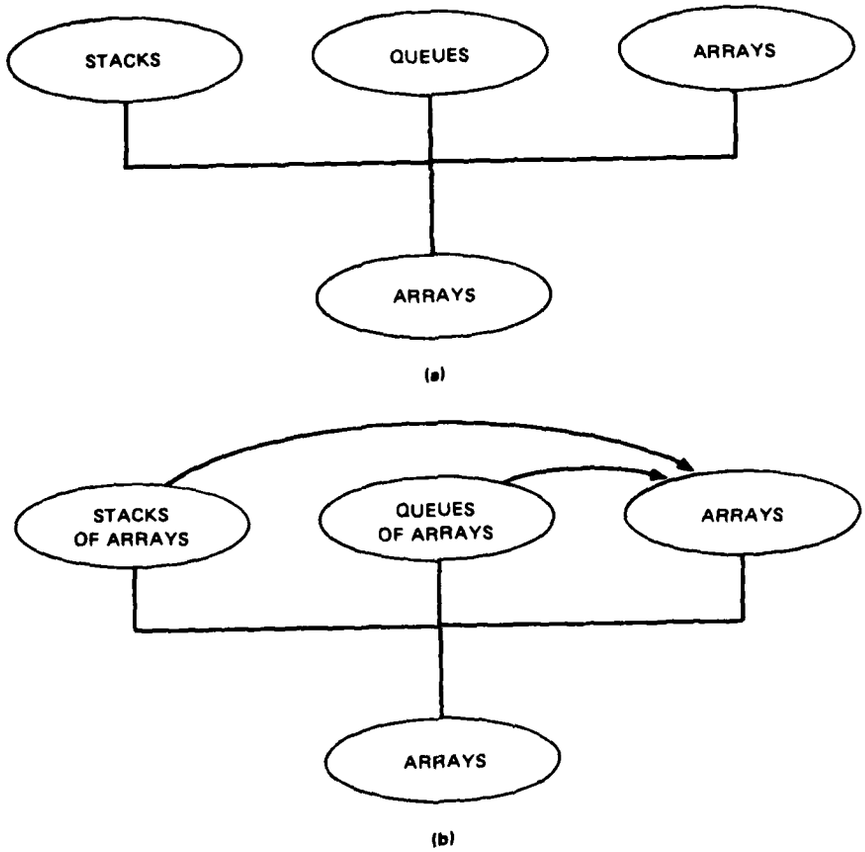


Figure IV-10: Two Actual Representation Clusters



4. Implementation Clusters

Just as we grouped upper and lower modules at representation time into units called representation clusters, we group upper and lower modules at implementation time into units called implementation clusters. As with a representation cluster, an implementation cluster consists of a set of upper modules, a set of lower modules, and a set of externally referenced modules (upper modules not implemented by the cluster but whose operations are invoked in the programs of the cluster).

The way in which external references are handled dictates whether or not the implementation cluster preserves module independence. Recall that if module A **externalref** module B, then some operations of module A are specified in terms of the operations and/or internal data structures of module B. This means that some operations of A access or change the internal data structures of B (and sometimes modules that are externally referenced by B). The implementation of A can access or change the internal data structures of B in either of two ways:

* By manipulating the lower-level data structures -- those that represent the internal data structures of B -- by invoking lower-level operations. This is how the implementations of B's operations access and change B's internal data structures.

* By calling operations of B to perform the changes.

Note that the first method does not preserve the independence of B's realization, because the programs of A now depend on B's representation. In addition, such a scheme would allow programs external to B to manipulate B's internal data structures in a potentially unauthorized way, thus violating the encapsulation provided by B. The second method preserves both the independence of B's realization and the encapsulation of B's internal data structures. Thus, the programs of an implementation cluster for a module A may invoke both the operations of the target machine and the operations of modules in any machine externally referenced by A.

In Section IV.B.2 it was stated that the **externalref** relation among modules must not be circular. The reason for this restriction is

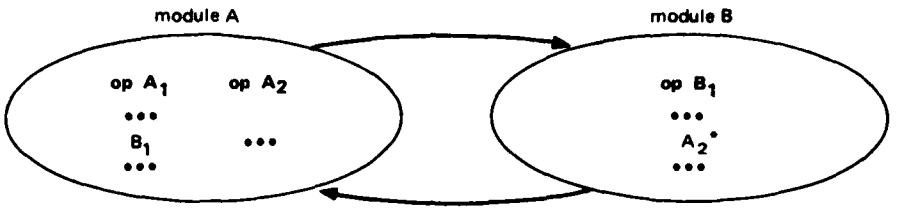
apparent after examining implementation clusters in which such circularity is allowed. For example, suppose A and B are modules, where A **externalref** B and B **externalref** A are both true. (See Figure IV-11 for a graphic representation of this example.) Suppose further that A1 and A2 are operations of A and that B1 is an operation of B. Now if the implementation of A1 invokes B1 and the implementation of B1 invokes A2, then the realizations of modules A and B are not independent, violating one of the fundamental rules of HDM. The lack of independence results from the fact that when the implementation of A1 invokes B1, the internal data structures of A are in some state known only to the implementor of A. This state may be inconsistent. Thus, when the implementation of B1 calls A2, the implementor of B would have to know about the state of A, in order to write a program that makes sense. Such knowledge would violate the independence of the realizations of modules A and B. For this reason, circularity in external references is not allowed. This restriction is somewhat stronger than need be, because circularity destroys realization independence only when there is circularity in the invocation of a single operation. However, having a hierarchical dependency of specifications is also desirable, allowing systems to be realized one module at a time.

5. Module Dependencies

To summarize the discussion of modularity, it is useful to present a schematic example containing modules, representation clusters, and implementation clusters (collectively referred to as clusters), and examine the interdependencies of the decisions in each cluster.

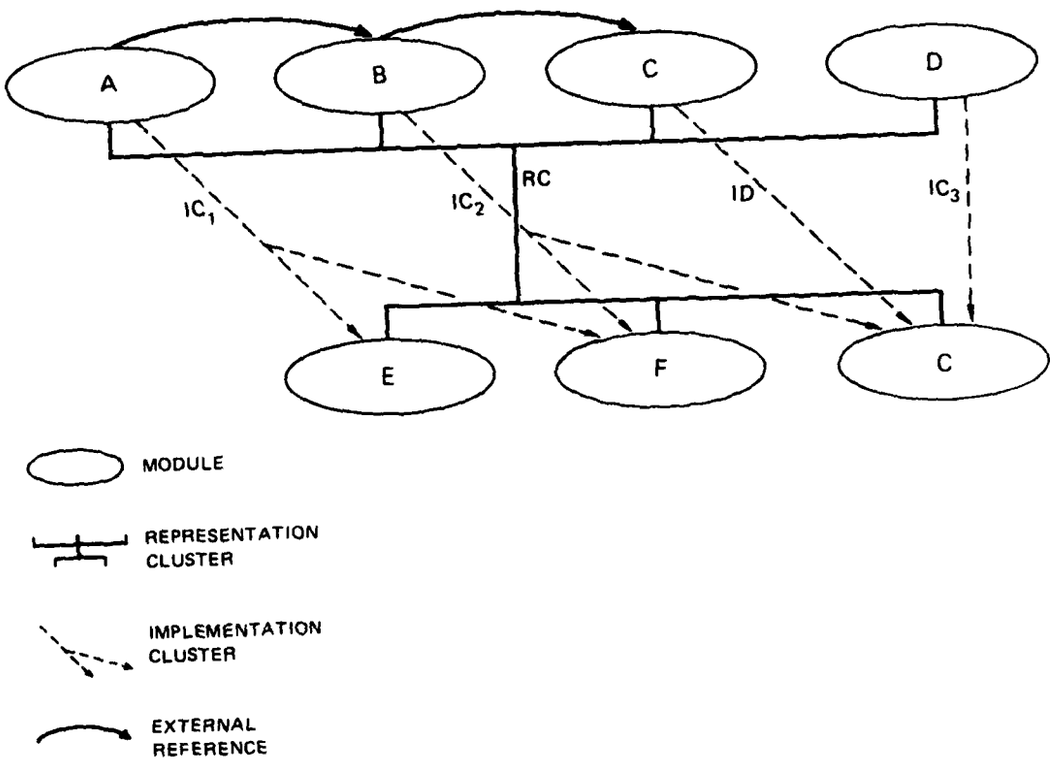
The example is a two-level system, described in Figure IV-12, with modules A, B, C, and D at the upper level and modules E, F, and C at the lower level. The relations A **externalref** B and B **externalref** C are true. There is one representation cluster, RC, with upper modules A, B, C, and D and lower modules E, F, and C. There are three implementation clusters: IC1 (upper module A, lower modules E and F), IC2 (upper module B, lower modules F and C), and IC3 (upper module D and lower module C). The upper appearance of module C is implemented identically by C at the lower level.

Figure IV-11: Inconsistent States in Circular External References



*Invocation of A_2 when module A is in an inconsistent state (known only to implementor of module A)

Figure IV-12: Modular Schema for an Example



Let us use the notation X **dep** Y1,...,Yn to mean that the decisions in cluster X may be dependent on the decisions in cluster Y1,...,Yn. Considering the **externalref** relation, we get the following: A **dep** B, C and B **dep** C. Note that **dep** for modules follows the **externalref+** relation. The **dep** relation is a worst-case approximation of dependencies. In the example, it may be the case that no decision in A is dependent on any decision in C (i.e., the set of decisions in B that A is dependent on may be disjoint from those in B that are dependent on any decisions in C). However, we do know that, for any clusters X and Y, if X **dep** Y does not hold, the clusters are totally independent.

From the representation cluster, we get the relation RC **dep** A, B, C, D, E, F. This results from the fact that a representation cluster is dependent on all of its upper and lower modules.

From the implementation clusters, we get the following relations: IC1 **dep** RC, A, B, C, E, F; IC2 **dep** RC, B, C, F; and IC3 **dep** RC, C, D. This results from the fact that an implementation cluster depends on all of its upper and lower modules, and on any representation clusters whose upper modules include any upper modules of the representation cluster.

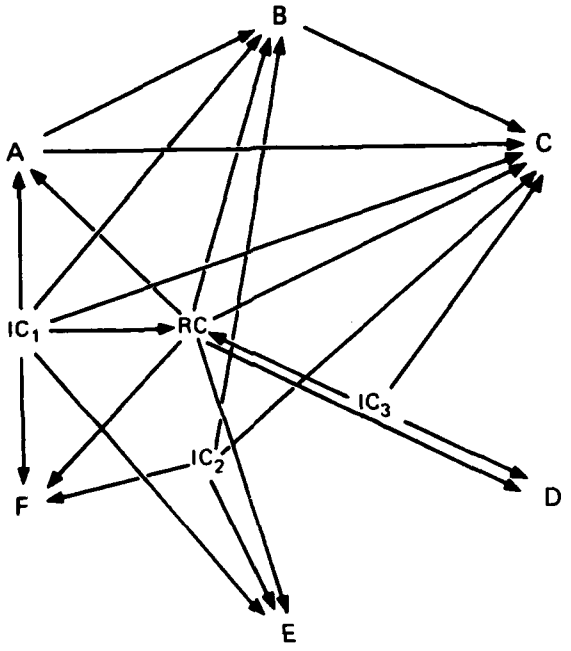
The total dependency graph is shown in *Figure IV-13*. The least dependent clusters are modules C, D, E, and F, and the most dependent clusters are implementation clusters IC1, IC2, and IC3. This is as it should be, because modules are where the decision-making process begins and implementation clusters are where the decision-making process ends. It may seem as if the example schema shows more dependency than should occur in a system. However, several observations are relevant.

1. The connectivity in actual systems is somewhat less than was presented in the example. Limiting both external references and the size of representation clusters cuts this connectivity considerably.

2. The dependency relation **dep** is a worst-case one, so the situation may in actuality be better than it appears.

3. In systems developed according to conventional methods, there may appear to be less connectivity (according to such measures as subroutine calling dependency). However, there is usually a great deal of implicit connectivity in such systems (with such items as shared data and common formats).

Figure IV-13: Dependency Graph for the Example



4. With this analysis, it is known that clusters for which no dependency relationship exists are totally independent. Knowledge of independence allows easy separation of work assignments once the system is decomposed into modular units.

The most important thing to remember is the close correspondence between the interdependency among HDM clusters and the interdependency among decisions in the software-development process.

6. Exceptional Conditions

In incorporating an exception mechanism into HDM, it was desired to use as simple a mechanism as possible that still allowed the benefits of the exception concept. The HDM exception mechanism operates only as a return from an operation invocation. No other signalling or control-changing mechanisms (e.g., transfers, state saving, or resuming of uninterrupted operations) are allowed. The specification of an operation under HDM adheres to the following rules:

* A single, "normal" return is associated with state change, a returned value, or both.

* An arbitrary number of "exceptional" returns can be named and associated with conditions (based on the arguments to the operation and the values of the internal data structures). Neither a state change nor a returned value is associated with an exceptional return.

These rules are supported in SPECIAL, HDM's language for module specification.

The following programming constructs facilitate the use and implementation of operations with this exception mechanism:

* A statement for signalling an exceptional condition in an implementation. This causes the program to return with notification of a particular exceptional condition.

* To allow for the handling of exceptional conditions, a statement that contains an operation invocation followed by a CASE-like construct, one case for each expected exceptional return.

These programming constructs are found in ILPL, HDM's abstract programming language. When ILPL programs are translated into actual running code, the above constructs can easily be expressed in terms of

the appropriate constructs of the target language.

V THE STAGES OF HDM

A. Introduction

The interdependency among decisions in software development prescribes a partial time ordering in making these decisions. At any given time, new decisions should be made that are dependent only on decisions that have already been made. As seen from the discussion of dependencies as related to modularity, the dependencies among groups of decisions correspond to dependencies among clusters in HDM. Thus, the dependency among decisions also prescribes a partial ordering in the generation of clusters in HDM.

However, the software development process is not so easy that this partial ordering can be adhered to in practice. The problem to be solved is not always fully understood throughout the software development process. As a result, any of several things may occur:

1. The dependency among decisions is not known, due to a lack of understanding, when certain decisions are made. This results in the violation of the partial ordering, sometimes with severe consequences in terms of work to be redone.

2. Wrong decisions are made due to a lack of understanding. When understanding finally occurs, much existing work may have to be redone.

In these cases the project manager must often choose between the cost of redoing the work and the cost of not redoing it. Many managers have chosen the false economy of not redoing work that had to be redone. These horror stories have led for some people to reject top-down design altogether and claim that in software development one should design two systems: the first to learn from (and throw away) and the second to be actually used.

This gap between the theory and practice of software development is a warning to those who put forth a step-by-step method of software development. However, it is desirable to put forth a suggested ordering of activities in software development, assuming ideal conditions. Even when things do not go perfectly and the time ordering is not strictly followed, the ordering provides a series of milestones by which to

measure progress. When the system has been completed, the results can be presented as if the system were developed in an orderly way. It is with these thoughts in mind that an ordering of activities in HDM, each activity called a stage, is presented.

It should be noted that the stages of HDM, which represent a structuring in time, should not be confused with the hierarchical and modular structure of the system, which is a static structuring of system components. The two concepts are orthogonal, and should be treated as such.

A decision should be recorded when it is made, even if that recording does not coincide with the ordering of the stages. The failure to record a decision has more severe consequences than making a decision in a non-standard order. Thus, it is sometimes desirable to depart from the stages as the decision-making process takes its own course.

The stages of HDM are as follows:

1. Conceptualization -- Identifying the problem to be solved and the role of the system in solving it.

2. External Interface Definition -- Defining the abstract machines (and possibly the abstract programs) that interact with the outside world. The modular structure of these abstract machines is also defined.

3. System Structure Definition -- Defining the hierarchy of abstract machines, and the decomposition of all machines into modules.

4. Module Specification -- Writing specifications (in SPECIAL) for each module of each machine in the hierarchy.

5. Data Representation -- Defining the internal data structures of each non-primitive abstract machine in terms of the internal data structures of the machine at the next lower level, using representation clusters.

6. Abstract Implementation -- Writing abstract programs to implement each operation of each non-primitive abstract machine, using implementation clusters.

7. Concrete Implementations -- Translating, by hand or machine, the abstract programs into executable code.

A graphic presentation of HDM's stages shown in Figure V-1. Each stage is discussed in some detail. This chapter concludes with a discussion of how the stages are actually used in the software development process. (Note that verification may be associated with several of these stages, notably 4, 6 and 7.)

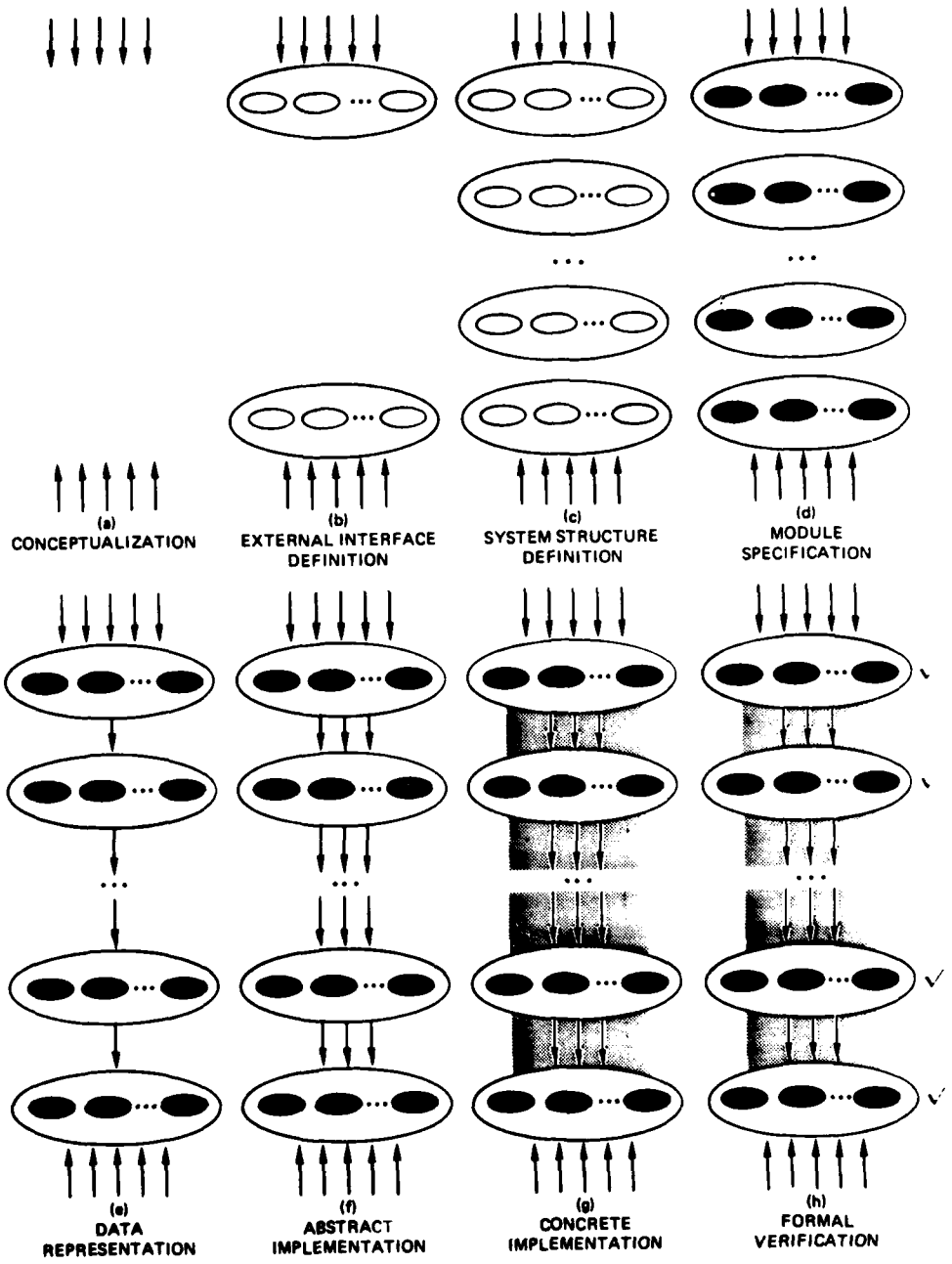
B. Conceptualization (Stage 1)

Conceptualization is the process of determining what problem the desired system should solve, without favoring any possible solution to the problem. This is sometimes called requirements analysis or systems analysis. Conceptualization is primarily a process of examining the world outside of the system and determining how the system fits into that world.

The idea in conceptualization is to put as few constraints on the system as possible, while still satisfying the desired requirements. When these constraints are determined, they should be stated as precisely as possible. Some work has been done in the formal statement of requirements of systems developed using HDM, particularly with respect to data security [5] (along with earlier work at the Mitre Corp.) and reliability in the presence of hardware faults [19]. Currently, HDM has no all-purpose language in which requirements can be formally expressed, so these requirements are typically stated by means of a mathematical model. In the near future, some attempt will be made to develop a general language for conceptualization.

In the conceptualization of systems developed according to HDM, it is important to relate the problem domain to the computational model of HDM. For example, some systems are best specified as an abstract machine specification (e.g., an operating system or a database manager), while others are best specified as an abstract program running on a particular abstract machine (such as a translator). In addition, some parts of the external world can also be modeled in terms of HDM mechanisms; for example, a network protocol can described as an abstract program, and a processor can be described as an abstract machine.

Figure V-1: The Stages of HDM



Note that requirements can be either extremely vague (e.g., a compiler must generate good code) or extremely specific (e.g., a specific hardware machine or network protocol must be used), thus putting widely differing constraints on the system designer. The conceptualization phase should therefore retain flexibility wherever possible, because there is no single level of abstraction at which all requirements can be stated. The more specific the requirement, the more detail that must be used to state it. It is the need for intellectual flexibility that makes conceptualization a difficult activity.

In terms of the decision model, requirements are the earliest decisions and affect the entire course of the development. Like other decisions, requirements may change during and after the development of the system. However, because they are established first, they are potentially the most difficult (or costly) to change. Every effort must be made to:

1. Establish as requirements only those decisions that are essential to the system development being successful.

2. Foresee the spectrum of changing requirements, so that the developed system will adhere to various sets of requirements within that spectrum, i.e., design as general a system as possible.

3. Structure the system so that any potential changes in requirements will have a minimal effect on the system, i.e., make the system easy to change with respect to that family of requirements.

C. Extreme Machine Definition (Stage 2)

In Stage 2 an informal definition of the highest level abstract machine (the system's external interface) and the lowest level abstract machine (the primitive, or target, machine) is produced. For each of these abstract machines, the informal definition consists of a decomposition into modules, an enumeration of the names of the data structures and operations for each module, and a description of the **externalref** relation for the modules at that level. Prose descriptions of the decompositions, as well as the names for the data structures and

operations, are also useful. In this stage, careful attention must be given to the results of the conceptualization stage; it is also useful to provide a justification that the proposed interface meets the requirements.

In Stage 2 decisions are made concerning the external behavior of the system, i.e., the behavior of the highest level abstract machine. These decisions are then grouped into modules such that the decisions within a module have high interdependence and the decisions in different modules have a low interdependence. The implementation of the top-level machine and the usability of the bottom-level machine must be carefully considered, since the function of the external interface may have to be redefined (if the requirements allow) as subsequent decisions are made. Unless the two levels share a module, the top and bottom machines are totally independent. Decisions within each level may possibly be related only if they occur within the same module or within modules connected by the **externalref+** relation.

D. Abstraction Formation and System Structure Definition (Stage 3)

In Stage 3 the activities are essentially the same as in the previous stage: forming the abstractions of machines at various levels, decomposing them into modules, listing the names of the operations and data structures of each module, describing the **externalref** relation among the modules, and writing informal descriptions of each operation of each module. However, the thought processes and decisions are those of design -- of constructing a system structure to connect to the external interface.

Two activities proceed simultaneously during this stage:

1. Intermediate Machine Definition -- This can proceed in any of several ways (as illustrated in Figure V-2): Top-down -- given an upper-level machine and a primitive machine, decide on a new machine somehow "closer" to the primitive machine than the upper-level machine to realize the upper-level machine; Bottom-up -- given the top-level machine and a lower-level machine, decide on a new machine somehow "closer" to the top-level machine than the lower-level machine that is easy to realize on the lower-level machine; and

Middle-outwards -- deciding on a good set of abstract mechanisms on which to build the system, and realizing both the mechanisms in terms of the primitive machine and the top-level machine in terms of the mechanisms. The method can be even more random than those stated above.

2. Modular Decomposition -- In decomposing a machine into modules, attention must be given to common facilities occurring at multiple levels. When maximum sharing of facilities at multiple levels occurs, it is possible in the subsequent implementation to write less code, to save space, and possibly to save time.

Note that as the system is developed in the subsequent stages, it may be desirable to add more intermediate levels or to change the modularization. These changes have minimal effect because they change neither the operations nor the data structures of existing levels.

In Stage 3 close attention is paid to how an abstract machine may be realized (abstractly) in terms of the machine below it in the hierarchy. "Ease of realization" should be a primary criterion in choosing intermediate abstract machines.

Decisions concerning different levels of the hierarchy are related only if the levels share a module. Decisions within a level are related only if the decisions occur within the same module or within modules connected by the **externalref+** relation.

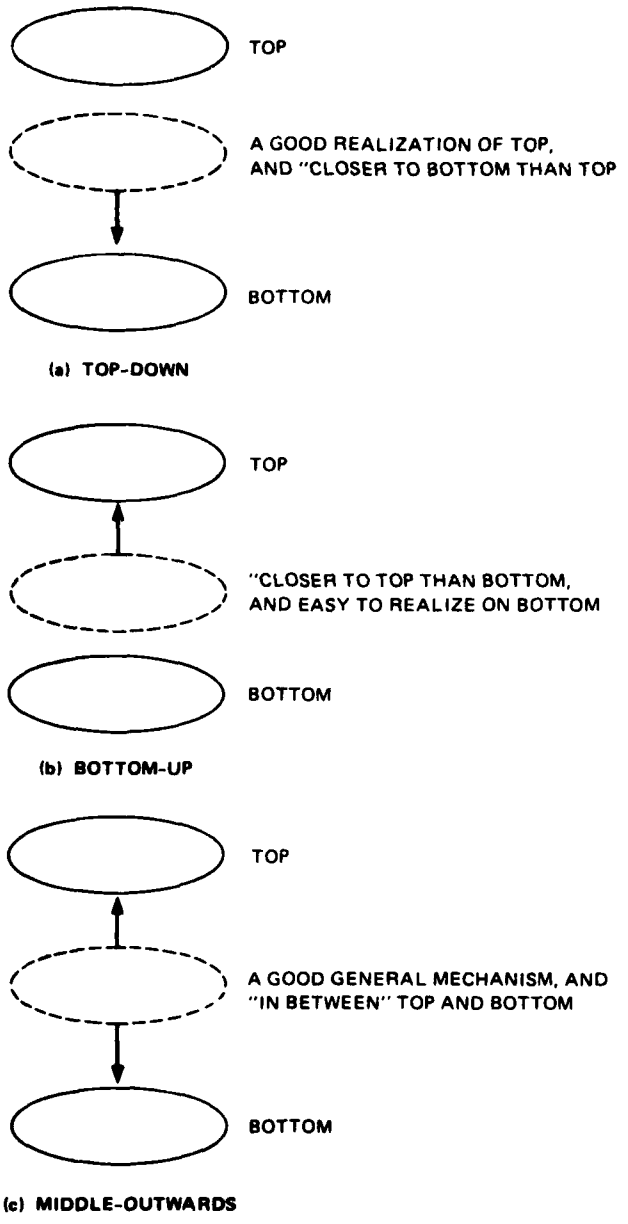
E. Module Specification (Stage 4)

In Stage 4 the specifications for each module of each abstract machine are written in SPECIAL. This provides a precise and explicit recording of decisions that were made during Stages 2 and 3. Specifying a module consists of:

1. Specifying initial values for each data structure.
2. Specifying each operation as a change in the values of the data structures.
3. Enumerating the operations and data structures to be shared between two modules for which **externalref** is true.

Module specification is listed as a separate stage because it takes a long time and because a lot is learned about the system. Many earlier

Figure V-2: Methods for Defining System Structure



decisions may have to be changed as a result of what is learned during this stage.

In this stage, careful attention must be given to completeness and consistency. A complete abstract machine specification defines the effects of each operation on all internal data structures of the machine. For a given operation this definition may specify specific changes for the data structures, or may specify a class of possible changes -- any one of which satisfies the specification. In the latter case the specification for the operation is nondeterministic.

To define consistency, it is useful to recall the definition of the state of an abstract machine: an association of a particular value with each element of the machine's data structures. An abstract machine specification is consistent if each operation specifies at least one valid destination state for every valid source state, without conflict. An example of an inconsistent specification is one which asserts two conflicting values (e.g., X = 2 AND X = 3). Clearly in no valid state can any data structure have more than one value, thus the inconsistency.

F. Data Representation (Stage 5)

In Stage 5 data representations are written for the data structures each of non-primitive abstract machine in terms of the data structures of the abstract machine at the next lower level. This consists of the following activities.

1. The modules of the upper and lower machines are divided into representation clusters.

2. For each representation cluster, (a) each internal data structure of each upper module is defined in terms of the internal data structures of the lower modules, and (b) invariants -- or consistency conditions -- on the internal data structures are written.

Data representations make it clear what the implementation of each upper-level operation must do to the lower-level data structures, but not necessarily how that is accomplished in terms of lower-level operations. The invariants express consistency conditions for the

lower-level data structures to represent a valid upper-level state (for example, in a module that maps keys to values, the keys may be stored in a particular order to facilitate searching).

Just like module specifications, data representations must also be complete and consistent. Here, completeness means that each data structure at the upper level must be given a value derivable from the values of the lower-level data structures. Consistency of a data representation means that for any given upper-level state, there is a unique set of lower-level states that represents it. An example of an inconsistent data representation is one which defines the values of two distinct upper-level data structures A and B as the value of the single lower-level data structure C. If the upper machine is ever in a state in which A and B have different values, an impossible situation arises. Clearly C cannot have two different values at the same time, and thus there is an inconsistency.

G. Abstract Implementation (Stage 6)

In Stage 6 each operation of each non-primitive abstract machine is implemented as an abstract program running on the abstract machine at the next lower level. This stage has two activities:

1. Dividing the set of upper modules into implementation clusters, with each cluster usually consisting of one upper module and a set of lower modules.

2. For each implementation cluster, (a) each operation is implemented as an abstract program in ILPL that calls operations of lower-level modules and of modules externally referenced by the module in which that operation occurs, and (b) an initialization program is provided.

Most of the decisions made at this stage are straightforward, because they have largely been dictated by decisions already made. If the implementation is extremely difficult, it means that insufficient care was made in making the earlier decisions.

In this stage, care must be taken to provide for the signalling of upper-level exceptions and the handling of lower-level ones. Any lower-level exception that is not handled is assumed not to occur. It

may also be the case that a given upper-level operation is not implementable using the operations of the lower level. In this case either the upper level or the lower level must be changed, to achieve a running system.

H. Concrete Implementation (Stage 7)

In Stage 7 the abstract programs (in ILPL) produced in Stage 6 are translated into code for which some compiler, assembler, or interpreter already exists. This translation can be done either by hand or automatically. The aim is to match the HDM computational model to the computational model of the target machine's language. Any discrepancy between computational models requires accommodation of the data types, the storage allocation scheme, and the signalling and handling of exceptions, as well as of the ability to encapsulate data.

After the concrete code for each operation of each implementation cluster has been produced, there is yet some work necessary to produce a running system. This requires a hierarchical initialization, starting with the lowest level and proceeding upward. When the top-level initialization program has been run, the system is ready to operate.

Research into ILPL translation and hierarchical initialization has just begun. It is anticipated that the results of this research will lead to changes to ILPL, and to automatic tools to accomplish both of these tasks. In addition, rewards may be reaped from optimization of the ultimate code, using information resulting from the application of HDM.

I. Formal Verification

Formal verification involves a mathematical proof of formally stated properties of a software system. In HDM, formal verification takes two forms:

1. Proofs that the user interface of the system has certain properties. These are called design proofs, and are associated with Stages 1, 2, 3 and 4.

2. Proof that the system's implementation (i.e., abstract programs) meets its specification. These are called implementation proofs, and are associated with Stages 5, 6, and 7.

Note that design proofs without implementation proofs do not guarantee that the system implementation has the desired properties.

Technologies for design proofs are conceptually straightforward. Examples for security are described in [5] and for fault tolerance in [19].

In general, the verification of a system's implementation is more difficult than the design proofs. HDM significantly enhances the possibility of verifying a large system as follows: by structuring the proof of the system according to the system's hierarchical and modular structure, and by allowing simpler assertions in terms of abstract data structures. Implementation proofs involve showing the consistency between code and specifications. Proofs also can be used to demonstrate (1) the consistency of modules and mapping functions and (2) that all exceptions are either handled or impossible. This is because implementation proofs for systems not having these desirable properties cannot be completed. The technology for implementation proofs is described in [17].

J. Remarks

In general, the (first seven) stages outlined above provide a rough framework for system development. It is stressed that the stages do not occur sequentially, and that considerable backtracking may in fact occur. Several acceptable variations to the stages outlined here are possible, as for example the following.

* Introduce new intermediate levels into the system after specifications for existing levels have already been written.

* Specify each machine as soon as it is identified, rather than waiting for the other machines to be defined.

* Determine the external interface after, rather than before, the intermediate levels (only if no specific constraints exist).

In all cases the representation of a level should follow its specification, and its implementation should follow its representation. Otherwise, the specifications of abstract machines and data representation would lose much of their value.

The clusters generated at each stage can become a point of dialogue and review among the system's designers and other parties. The result of this dialogue and review may often be that early decisions are discarded and replaced, resulting in work that must be redone. However, this is a normal procedure in a system development. If this process is stifled, inferior decisions will be allowed to perpetuate, resulting in an inferior product that may not meet its requirements. The stages of HDM provide for many natural review points in the software development process, with the aim of correcting inferior decisions as early as possible in the software development process.

VI THE EFFECTIVE USE OF HDM

A. Introduction

The preceding chapters of this volume have presented the concepts and mechanisms of HDM, but have not dealt with how to use HDM effectively. Software development is a difficult undertaking, regardless of what aids are used. HDM is primarily an aid to structuring and recording the decisions made, but it cannot make these decisions for the software developer.

A software developer educated in conventional techniques who is just beginning to use HDM will have an even more difficult time than he would using conventional techniques, because HDM has a particular computational model that places many constraints on the software developer. For example, for a particular problem, a developer may have in mind a technique that violates the restrictions of HDM, and he will be forced to find another way to solve it.

This chapter provides guidelines on how to use the concepts and mechanisms of HDM to develop software systems, both to provide heuristics as to how to make decisions and to help a software developer in becoming accustomed to a new way of thinking.

Most of the guidelines listed here address the early decisions and their structuring (i.e., hierarchical and modular decomposition). Later decisions concerning specification, representation, and implementation are illustrated in Volume III. This chapter concludes with a statement of trade-offs and a list of potential indications of misuse of HDM.

B. The Use of Abstraction

The most important step in the use of HDM is to learn how to use abstraction effectively, particularly abstract machines and abstract programs.

A good exercise is to imagine a particular programming problem and to postulate an abstract machine (and possibly an abstract programs) to

solve it. It is surprising how many problems yield machines of the following forms:

* A machine that manages a collection of abstract objects, where most often an object is a simple data structure. The rules for manipulating such objects will be enforced by defining a restricted set of operations. Details of representation, implementation algorithms, and allocation strategies are not part of the machine's definition.

* A content-addressed memory, or a machine that performs a table look-up (e.g., a directory).

* A machine that performs certain kinds of data transformation that are easily described in terms of the end results (e.g., sorting).

* A machine that perpetrates some kind of "illusion" on its users (see below).

In postulating abstract machines, there must be a great emphasis on the nature of the data and on the primitive operations on the data, rather than on the actual processing algorithm for the data. For example, if the task at hand is to do process control, one first asks not what the steps in process control are, but what is the data that must be manipulated in order to perform process control functions. The algorithm can always be developed and investigated once the primitive machine is decided upon.

Some of the "standard" abstractions that have proven themselves useful in software systems are

* A virtual processor (achieved through multiprogramming) that allows multiple programs -- each executing in a separate process -- to operate as if each one had a hardware machine to itself. Each process, however, has access to the central processor for relatively small slices of time.

* A virtual memory, which provides a program with the illusion of a large, directly-addressable memory space, even if the actual information is scattered throughout primary and second memories that are shared by many users. In the actual implementation, a program generates a virtual address that either is mapped via a memory mapping table (if the information is in core) or generates a memory fault, causing the information to be brought into core from secondary storage.

* A list-processing language with automatic garbage collection (e.g., LISP) in which a user gets the illusion that he has an unbounded amount of list storage available. The amount of list storage available to the user is actually finite, but list cells that are no longer in use are automatically reclaimed and presented to the user as "new" list cells. A user can write programs that have no knowledge of this reclamation mechanism.

* Relational data bases, in which a user can access data based solely on certain properties, or relations, on both the structure and content of the data. The actual file structure and lookup mechanism is hidden from the user. A relational data base is an example of an abstract machine that was proposed without a particular realization in mind. The efficient realization of relational data bases efficiently is still an open research problem.

Note that all of these standard abstractions are especially interesting because they deceive the user (as in idea (5) above). A prospective user of HDM should become familiar with "standard" abstractions so as to be better able to create his own.

Two questions should be asked immediately after defining an abstract machine, in order to see whether the abstract machine is appropriate for the task at hand:

1. Is it easy to use?

2. Is it easy to implement?

Both the intended use and the intended implementation of the abstract machine should be sketched out, informally at least, to evaluate the choice of abstraction. In many cases, small changes that improve the abstraction will become obvious. In other cases, a completely different abstraction will be needed, or even a hierarchy of abstractions Section .

C. Modularity

Once an abstract machine has been decided upon, it is necessary to break it up into modules, or units of specification that can be implemented separately.

In decomposing an abstract machine into modules, the operations and

data structures of the machine should be examined carefully. The combination of the two following conditions is that under which a group of data structures and operations can be found such that the group forms a separate module without external references.

1. No operation within the group accesses any data structures outside of the group, and
2. No operation outside of the group accesses any data structures within the group.

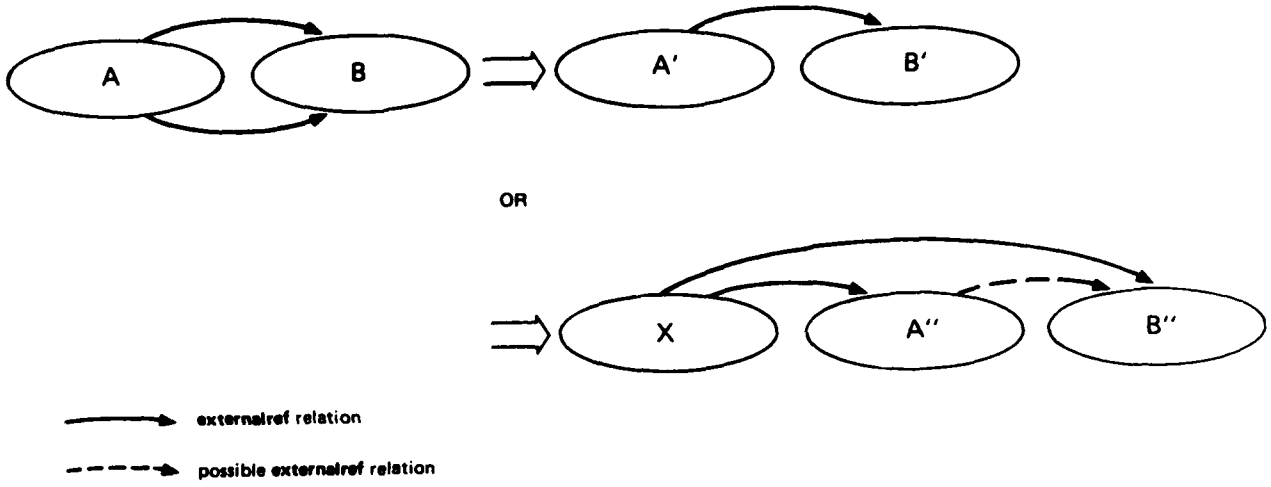
In many cases these constraints cannot be totally followed, but a group of operations and data structures almost follows these rules. In this case a module can be formed, in which the exceptions to the rules determine the external references to and from the module. Whether or not all such cases should be decomposed depends on the number and nature of the external references: there should not be "too many" external references, i.e., the module should be loosely coupled to the rest of the machine; and the **externalref** relation should be non-symmetric (i.e., **externalref+** should not contain circularities). This process continues until a satisfactory modular decomposition is reached.

There may be some difficulty in achieving the non-symmetry in the **externalref** relation. For example, suppose there are two potential modules A and B that externally reference each other. This problem can often be solved by (1) moving operations from one module to another so that the external references go in one direction only or (2) creating a third module containing operations that externally reference each other (see Figure VI-1). If even this method fails, then A and B must be combined into a single module.

D. Hierarchical Decomposition

This is the process of developing a sequence of intermediate machines to bridge the gap between the top and bottom machines in a system. Proceeding from bottom to top, the sequence should describe a gradual evolution from the facilities that are present to the facilities that are desired. This means that certain facilities appear and others are hidden as one proceeds upward on the hierarchy. In most cases, when

Figure VI-1: Achieving **externalref** Non-symmetry



two related abstractions are considered, it is obvious which one belongs "above" the other in the hierarchy. However, this is not always so, when it seems that two abstractions are interdependent. In this case an approach called sandwiching can be applied, in which two slightly different manifestations of the same abstraction appear at two different levels, surrounding the abstraction with which there is interdependence. (See Figure VI-2a.)

For example, it may appear that a virtual memory system (called VM) and a multiprogramming system (called processes) are interdependent for purposes of realization: VM needs processes to enable a process to wait for a requested memory block to be brought into main memory, and processes need VM to store the state information for the large number of processes that it manages. At the bottom would be a physical storage facility (called fixed memory). This problem can be solved by a sandwich with an intermediate facility, a multiprogramming facility in which process states are stored in physical storage (called fixed processes). The structure of the resulting hierarchical system is shown in Figure VI-2b. Fixed processes is realized in terms of fixed memory, VM is realized in terms of fixed processes and fixed memory, and processes is realized in terms of fixed processes and virtual memory. Note that fixed memory and fixed processes are hidden, and VM and processes are introduced, as one proceeds from bottom to top.

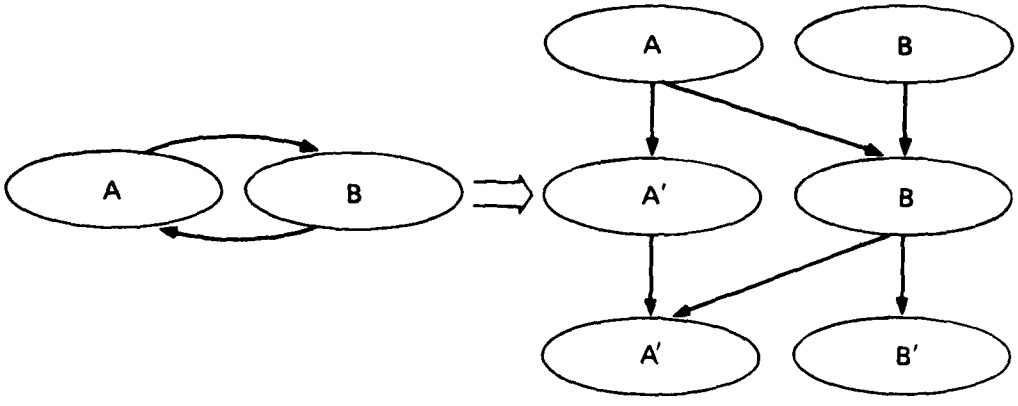
E. Data Representation

There is only one recommendation for decomposing a data representation between two abstract machines into representation clusters: allow representation clusters with multiple upper modules only when necessary. Two schemas for this recommendation present themselves:

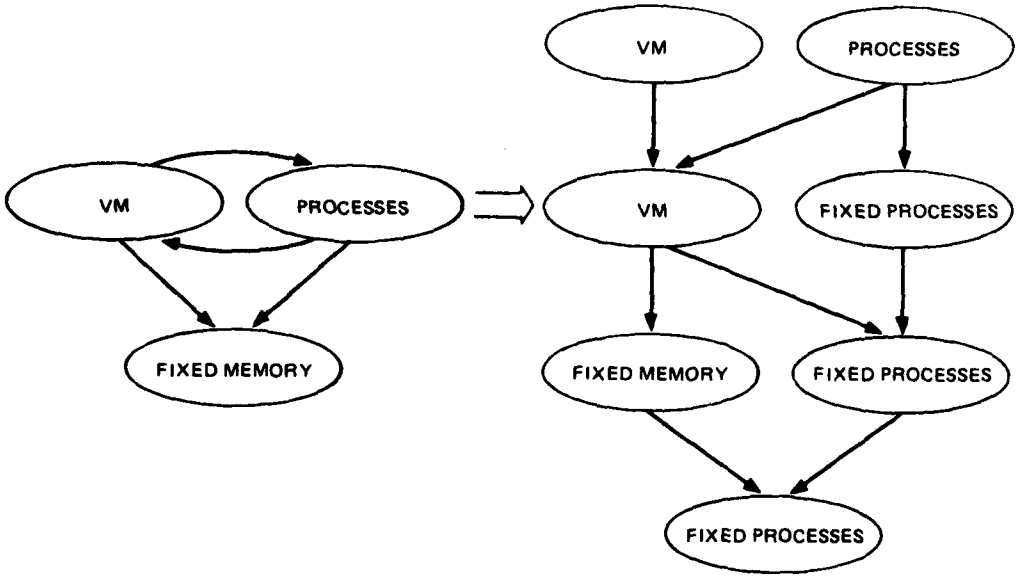
1. A single representation cluster with two or more upper modules may be split up when the data structures of the upper module map to disjoint data structures in the lower module (illustrated in Figure VI-3a).

2. When external references among upper modules occur, the upper modules do not always have to appear in the same

Figure VI-2: Sandwiching in Hierarchical Decomposition



(a) SCHEMATIC REPRESENTATION (arcs represent realization dependency)



(b) EXAMPLE FROM OPERATING SYSTEMS (arcs represent realization dependency)

representation cluster (illustrated in Figure VI-3b).

F. Indications of Misuse of HDM

There are several potential situations that may indicate a misuse of HDM. Following is a list of some of the most common ones, along with suggestions as to the possible underlying difficulty and and proposed solutions.

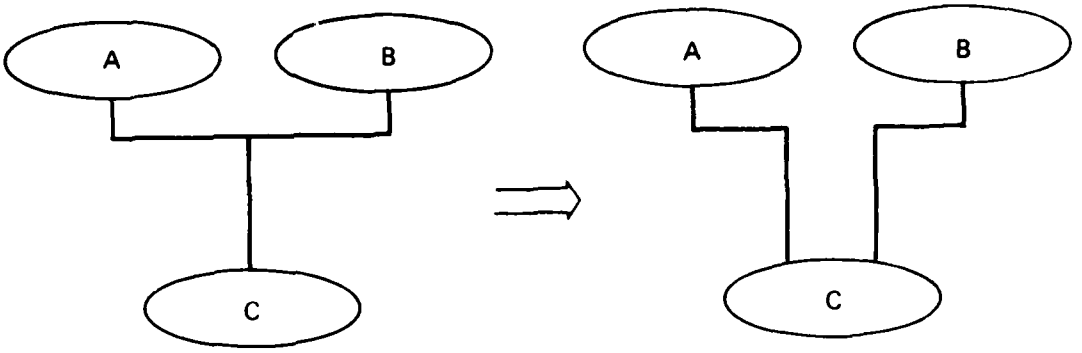
* Too many interconnections (external references) among modules at the same level. In this case the modules are not really as independent as they should be. This difficulty can be caused either by having too many modules - in which case multiple modules should be combined - or by having the wrong decomposition - in which case a better decomposition (according to some of the criteria mentioned above) would help.

* Module specifications that are too complex, e.g., too long, having too much detail, or having too many primitives. This problem is caused most often by thinking at too low a level of abstraction. One should try to eliminate the irrelevant details or rethink the "essence" of the problem to be solved. If the number of functions alone is the problem, then one might try further modular decomposition or a different choice of "primitive" operations (they may be too primitive for the task at hand). One may have designed too much generality into the system. Unless wide application of the module is a certainty, a particularization of the functionality of the module to the problem at hand may help.

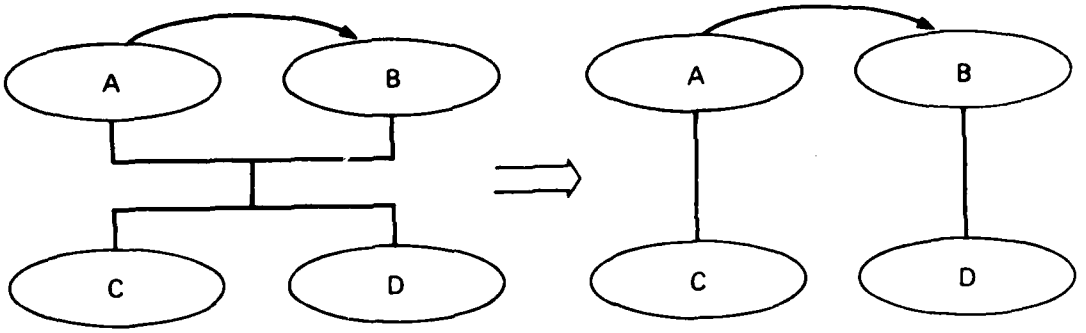
* Representation clusters or implementation clusters that are too complex. This can be caused either by having made the wrong representation or implementation decisions, or (more probably) by having too great a jump in abstraction between the upper and lower machines. In the latter case, the insertion of one or more intermediate levels or a different choice of upper and lower machines is helpful.

* Difficulty in visualizing the solution of the problem using abstract machines. This is often the result of a designer having dealt only with programs in the past, and not yet having acquired the experience to think in terms of abstract machines. However, there are certain classes of problems for which the abstract machine model is not attractive (e.g., applications that require a lot of processing but have very little data, such as numerical mathematics). In this case other techniques, such as procedure abstraction, can still be used with HDM.

Figure VI-3: Separation of Representation Clusters



(a) SHARED REPRESENTATIONS



(b) EXTERNAL REFERENCES

AD-A091 270 SRI INTERNATIONAL MENLO PARK CA F/G 9/2
THE SRI INTERNATIONAL HIERARCHICAL DEVELOPMENT HANDBOOK, VOLUME--ETC(U)
JUN 79 W L SUTTON, L ROBINSON N00123-76-C-0195
UNCLASSIFIED NOSC-TD-366-VOL-1 NL

* Having too many modules or too many levels. This is not always a difficulty, because HDM was intended to allow maximum structuring to ensure certain benefits in the development of large software systems. The larger the system, the more structure that will be expected, and indeed required, to make the effort manageable. However, too much structure can always be reversed without any creative effort: multiple modules can always be combined into a single one, and intermediate levels can always be eliminated.

All determinations of difficulty are of course subjective. In many cases HDM will be blamed for needlessly inserting complexity into a system. Another way of looking at it is that HDM merely reflects the complexity that is there already. One can judge this better by reading the example.

G. Conclusion

Using these guidelines successfully still takes considerable skill. However, if a system developed according to HDM meets all the guidelines without encountering these difficulties, then it is reasonably certain that the concepts and mechanisms of HDM have been successfully applied.

REFERENCES

[1] Boyer, Robert S. and J Strother Moore.
A Formal Semantics for the SRI Hierarchical Program Design Methodology.
Technical Report, SRI Computer Science Laboratory, November 1978.

[2] E. W. Dijkstra. "Notes on Structured Programming," O. J. Dahl, et al (ed.), Structured Programming, Academic Press, 1972.

[3] Feiertag, R. J. and P. G. Neumann.
The Foundations of a Provably Secure Operating System (PSOS), pages 115-120.
Proc. NCC, June, 1979.

[4] Feiertag, R. J. and K. N. Levitt.
RTOS.
Technical Report, SRI International, April 1979.
Final Report.

[5] Feiertag, R. J., K. N. Levitt, and L. Robinson.
Proving Multilevel Security of a System Design.
Operating Systems Review 11(5):57-66, November 1977.
Proc. Sixth Symposium on Operating System Principles, Purdue University, West Lafayette, Indiana.

[6] Floyd, R. W.
Assigning Meanings to Programs.
Mathematical Aspects of Computer Science, Vol. 19, Proc. of Symposia in Applied Mathematics.
American Mathematical Society, Providence, Rhode Island, 1967, pp. 19-32.

[7] Hoare, C. A. R. "Notes on Data Structuring," O. J. Dahl, et al (ed.), Structured Programming, Academic Press, 1972.

[8] Hoare, C. A. R.
Proof of Correctness of Data Representations.
Acta Informatica 1:271-281, 1972.

[9] McCauley, E. J. and P. Drongowski.
KSOS: Design of a Secure Operating System.
Proc. NCC, June, 1979.

[10] Neumann, P. G., R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson.
A Provably Secure Operating System: The System, Its Applications, and Proofs.
Technical Report, Stanford Research Institute, February 1977.
Final Report, SRI Project 4332.

[11] Neumann, P. G.
Computer System Security Evaluation, pages 1987-1095.
Proc. NCC, January, 1978.

[12] Parnas, D. L.
On the Criteria to Be Used in Decomposing Systems into Modules.
Communications of the ACM 15(12):1053-1058, December 1972.

[13] Parnas, D.L.
On the Design and Development of Program Families.
IEEE Trans. on Software Engineering 2(1):1-9, March 1976.

[14] Parnas, D. L.

A Technique for Software Module Specification with Examples.
Communications of the ACM 15(5):330-336, May 1972.

[15] Price, W. R.
Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems.
PhD thesis, Carnegie-Mellon University, Department of Computer Science, June, 1973.

[16] Robinson, L.
HDM — Command and Staff Manual.
Technical Report CSL-49, SRI International, February 1978.
SRI Project 4828.

[17] Robinson, L. and K. N. Levitt.
Proof Techniques for Hierarchically Structured Programs.
Communications of the ACM 20(4):271-283, April 1977.

[18] Roubine, O. and L. Robinson.
The SPECIAL Reference Manual.
Technical Report CSL-45, Stanford Research Institute, January 1977.
SRI Project 4828.

[19] Wensley, J. H., L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock.
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control, pages 1240-1255.
Proc. IEEE Vol. 66, No. 10, October, 1978.